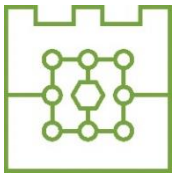
Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji



Imię i nazwisko: Daria Pyk

Numer albumu: 149618

Zabezpieczenia formularzy przeciwko najpopularniejszym atakom z listy OWASP Top 10 w aplikacjach internetowych

Forms security against the most popular attack on the list of OWASP Top 10 in the web applications

Praca magisterska

na kierunku: Informatyka

specjalność: Cyberbezpieczeństwo

Praca wykonana pod kierunkiem: **dr Radosław Kycia**

Kraków, 2024

Streszczenie

W ramach niniejszej pracy przetestowano zabezpieczenia prostej aplikacji typu ToDo List przed wybranymi podatnościami z listy OWASP Top 10, takimi jak: Server-Site Request Forgery (SSRF), Cross-Site Scripting (XSS), HTTP Parameter Pollution (HPP) oraz SQL Injection. Stworzono prostą aplikację formularzową zawierającą powyższe podatności, a następnie przeprowadzono ataki przy użyciu narzędzi takich jak: ZAP oraz Postman. Następnie w ramach części badawczej pracy zastosowano różne strategie zabezpieczeń, takie jak: filtrowanie danych, parametryzacje zapytań, zastosowanie odpowiednich bibliotek, takich jak: ”DOMPurify” oraz techniki ORM. Implementacja zabezpieczeń opierała się również na mechanizmie podwójnej weryfikacji użytkownika oraz stosowaniu ”białych list”. Wyniki jednoznacznie wykazały, że zaimplementowane metody są skuteczną formą zabezpieczeń przed tymi atakami i w dużym stopniu poprawiają poziom bezpieczeństwa aplikacji.

Słowa kluczowe: aplikacja formularzowa, OWASP Top 10, XSS, SQL Injection, HPP, SSRF

Abstract

This work tested the security of a simple ToDo List application against selected vulnerabilities from the OWASP Top 10 list, such as Server-Site Request Forgery (SSRF), Cross-Site Scripting (XSS), HTTP Parameter Pollution (HPP) and SQL Injection. A simple form application containing the vulnerabilities above was created, and then attacks were carried out using tools such as ZAP and Postman. The research work then applied various security strategies, such as data filtering, query parameterization, the use of relevant libraries such as: "DOMPurify" and ORM techniques. The implementation of security was also based on the mechanism of double user verification and the use of "white lists." The results clearly showed that the implemented methods are an effective form of protection against these attacks and greatly improve the security level of the application.

Keywords: form application, OWASP Top 10, XSS, SQL Injection, HPP, SSRF.

Spis skrótów i oznaczeń

XSS - Cross- Site Scripting – wywołanie skryptów krzyżowych
CSRF – Cross-Site Request Forgery - fałszywe żądania między witrynowe
SSRF – Server-Side Request Forgery - fałszywe żądania po stronie serwera
HTTP - HyperText Transfer Protocol - Protokół Transferu Hipertekstu
HPP – HTTP Parameter Pollution – zanieczyszczenie parametrów HTTP
ORM – Object Relational Mapping – mapowanie obiektowo-relacyjne
SQL - Structured Query Language - Język zapytań strukturalnych
OWASP - Open Web Application Security Project - Otwarty Projekt Bezpieczeństwa Aplikacji Sieciowych
DOM - Document Object Model - Model Obiektowy Dokumentu
URL - Uniform Resource Locator - Jednolity Lokalizator Zasobów
CSP - Content Security Policy - Polityka Bezpieczeństwa Zasobów
JSX - JavaScript XML - JavaScript Rozszerzony o XML
SPA - Single Page Application - Aplikacja Jednostronicowa
HTML - Hypertext Markup Language - Hipertekstowy Język Znaczników
JSON - JavaScript Object Notation - JavaScript Notacja Obiektów
JWT - JSON Web Token - Token Sieci Web w formacie JSON
HMAC - Keyed-Hash Message Authentication Code - Kod Uwierzytelniania Wiadomości z Kluczem
SHA - Secure Hash Algorithm - Bezpieczny Algorytm Skrótu

HTTPS - Hypertext Transfer Protocol Secure - Bezpieczny Protokół Transferu Hipertekstu
API - Application Programming Interface - interfejs programistyczny aplikacji, który umożliwia komunikację pomiędzy różnymi systemami oprogramowania
REGEXP - Regular expressions - wyrażenia regularne

Spis treści

1. Wprowadzenie

1.1 Cel pracy

Celem niniejszej pracy jest przetestowanie zabezpieczeń aplikacji typu ToDo List przed podatnościami znajdującymi się na liście OWASP Top10. W ramach badań skoncentrowano się na wybranych podatnościach, takich jak: wstrzyknięcie kodu SQL, XS, SSRF oraz HPP, które są dostosowane do wybranego typu aplikacji formularzowej.

1.2 Zakres pracy

Zakres pracy obejmuje stworzenie prostej aplikacji formularzowej ToDo List zaprojektowanej w taki sposób, aby podatna była na typowe zagrożenia bezpieczeństwa, które sklasyfikowane są w liście OWASP Top 10. Podczas pracy skupiono się na przeprowadzeniu wybranych ataków, takich jak: wstrzyknięcie kodu SQL, XSS, SSRF oraz HPP. W tracie testów wykorzystano popularne narzędzia pentesterskie, takie jak: OwaspZAP oraz Postman. Następnie przy użyciu odpowiednich narzędzi i bibliotek, takich jak: ORM, biblioteka "DOMPurify", parametryzacja zapytań, zabezpieczono aplikację w sposób uniemożliwiający ponowne przeprowadzenie powyższych ataków.

1.3 Metodyka pracy

W pracy wykorzystano metodykę wytwarzania oprogramowania "Waterfall", która jest uznawana za klasyczną metodę liniową. Skupia się ona na sekwencyjnym rozwijaniu projektu. Część praktyczna pracy została podzielona na kilka etapów, a każdy kolejny rozpoczynano dopiero po pomyślnym zakończeniu poprzedniego. Początkowo skupiono się na stworzeniu podstawowej aplikacji formularzowej, posiadającej podatności, a następnie wprowadzono zabezpieczenia w celu ich likwidacji. Skuteczność wprowadzonych zabezpieczeń była weryfikowana przy pomocy przeprowadzeniu odpowiednich testów. Wyżej wspomniany rodzaj metodyki jest idealny podczas samodzielnego tworzenia oprogramowania. Podczas testowania aplikacji wykorzystano podejście "White-box Testing" (test białej skrzynki). Jego wybór wynikał ze znajomości wewnętrznej struktury aplikacji, co pozwoliło na dobór właściwych testów.

1.4 Motywacja

W dzisiejszych czasach dostęp do Internetu jest powszechnym narzędziem, które znacznie ułatwia oraz usprawnia wiele czynności dnia codziennego. Dzięki dynamicznemu rozwojowi aplikacji internetowych, w łatwy sposób możemy rozwiązać wiele spraw urzędowych oraz prywatnych. Niemniej jednak rozwój ten przekłada się na zwiększoną liczbę cyberprzestępstw, na które narażeni są użytkownicy. Wiele organizacji i firm, takich jak: banki, hotele czy nawet sklepy, posiadają w swoich witrynach formularze, które umożliwiają interakcje z klientami i wprowadzanie danych. Często uzupełnianie takich pól wiąże się z wprowadzeniem różnego rodzaju wrażliwych informacji, takich jak: dane osobowe, adresowe lub dane dotyczące kart płatniczych. Pola te mogą posłużyć jako potencjalny wektor ataku. Dlatego tak ważne jest zaimplementowanie przez twórców różnego rodzaju mechanizmów ochrony oprogramowania, które uniemożliwi ich skuteczne przeprowadzenie. Skutkami powodzenia tego typu działań może być, np.: naruszenie integralności aplikacji, uszkodzenie środowiska uruchomieniowego, jak również wykradnięcie poufnych danych użytkowników. Do najczęstszych rodzajów ataków, na aplikacja formularzowe możemy zaliczyć ataki tj.: wstrzyknięcia kodu SQL, XSS oraz SSRF. Obecnie zapewnienie ochrony przed tego rodzaju zagrożeniami jest ułatwione dzięki dostawcom narzędzi programistycznych, takim jak biblioteki, freameworki, języki programowania oraz inne platformy wykorzystywane do tworzenia oprogramowania, które posiadają wbudowane mechanizmy ochronne. Szczególną uwagę należy zwrócić na zmasowane ataki botów, które stanowią poważne zagrożenie dla aplikacji internetowych. Wraz z rozwojem sztucznej inteligencji ich twórcy implementują elementy uczenia maszynowego, które w znaczący sposób przyczyniają się do zwiększenia skuteczności ataków przeprowadzanych przy użyciu botów. Ciągły rozwój oprogramowania może skutkować powstaniem nowych podatności i ataków, które w istotny sposób mogą narazić bezpieczeństwo oprogramowania. Dlatego tak ważna jest implementacja odpowiednich zabezpieczeń, które w odpowiedni sposób będą chronić oprogramowanie oraz jej użytkowników

Część teoretyczna

2. Opis wybranych zagrożeń z listy OWASP Top 10 w kontekście bezpieczeństwa formularzy w aplikacjach internetowych

2.1 Czym są aplikacje formularzowe?

Aplikacje formularzowe są to aplikacje internetowe lub mobilne, które opierają się głównie na interakcji z użytkownikiem, która odbywa się za pośrednictwem różnego rodzaju pól, przycisków oraz innych elementów służących do wprowadzania danych przez użytkownika. Rozwiązania te mogą być wykorzystane między innymi w:

- formularzach rejestracyjnych,
- formularzach kontaktowych,
- formularzach zamówień lub rejestracji na wydarzenia,
- formularzach ankietowych lub opinii,
- formularzach płatności.

Jedną z popularnych aplikacji formularzowych jest "Google Forms", która została stworzona przez firmę Google. Umożliwia ona tworzenie ankiet online, dzięki którym użytkownicy mogą pozyskiwać różnego rodzaju informacje. Rozwiązanie to jest szeroko stosowane przez firmy, które zajmują się między innymi badaniem trendów rynkowych, ponieważ umożliwia ono, w łatwy sposób dotrzeć do szerokiego grona odbiorców. Tego rodzaju rozwiązania posiadają wiele zalet, takich jak redukcja kosztów związanych ze sposobem pozyskania danych oraz w znaczący sposób przyśpieszają proces pozyskiwania informacji. Kolejną firmą wykorzystującą to rozwiązanie jest PayPal, który ogromną rolę odgrywa między innymi, w sklepach internetowych oraz sektorze e-commerce, ułatwiając międzynarodową sprzedaż wielowalutową. Jest to platforma, która umożliwia użytkownikom przesyłanie oraz odbieranie pieniędzy przez Internet. Rozwiązanie to pozwala firmom rozwijać swoją sprzedaż, a osobom fizycznym ułatwia dostęp do zakupów i znacznie go przyśpiesza. Obecnie aplikacje formularzowe są niezwykle pomocnym narzędziem, które nie tylko pozytywnie wpływa na efektywność pracy, ale pozwala również na automatyzacje wielu procesów zachodzących w firmach. Szczególną rolę, w kwestii komunikacji z klientem odgrywają formularze kontaktowe, które wiele firm implementuje na swoich stronach. Rozwiązanie to umożliwia bezpośredni kontakt z firmą, możliwość zadawania pytań oraz możliwość zgłaszania problemów bezpośrednio do administratora. Niemniej jednak, każdy formularz powinien być w odpowiedni sposób zabezpieczony, gdyś może stać się potencjalnym wektorem ataku. Zagrożeń wynikających z niewłaściwego zabezpieczenia formularzy może być

wiele i są może być to na przykład: kradzież danych, manipulacja zawartością strony, a nawet uzyskanie dostępu do bazy danych. Dane wprowadzane do pól powinny być zabezpieczane i walidowane w sposób, który uniemożliwi potencjalny atak, prowadzący do kradzieży wrażliwych danych. Formularze nie tylko ułatwiają komunikację z potencjalnymi klientami aplikacji, ale też służą do zbierania informacji, które użytkownik dostarcza za pośrednictwem ich pól, dlatego ochrona tych danych jest niezwykle istotna.

2.2 OWASP

OWASP (Open Web Application Security Project) [1] to międzynarodowa organizacja, non profit, która skupia się na poprawie bezpieczeństwa oprogramowania. Fundacja zrzesza dziesiątki tysięcy osób, a jej oddziały znajdują się niemal na całym świecie. OWASP zajmuje się tworzeniem narzędzi, dokumentacji oraz projektów, które mają umożliwić organizacjom tworzenie i rozwijanie bezpiecznych aplikacji. Na platformie Github, społeczność udostępniła otwarty projekt - WebGoat [2], który stworzony został jako narzędzie do nauki zagrożeń bezpieczeństwa aplikacji internetowych. Dzięki niej użytkownicy mogą testować luki występujące w aplikacjach, a przy tym nabyć praktyczne doświadczenie w odkrywaniu ich podatności. WebGoat oferuje obsługę aktualnie popularnych podatności i wyjaśnia zagrożenia oraz skutki ataków. Organizacja opracowała również dokument, który charakteryzuje listę dziesięciu najważniejszych zagrożeń dla bezpieczeństwa aplikacji internetowych. Członkowie fundacji tworzą różnorodne dokumenty, które zawierają szczegółowy opis klasyfikacji zagrożeń dotyczących różnych technologii. Lista OWASP Top 10 [3] aktualizowana jest regularnie, w miarę ewolucji zagrożeń, które powszechnie występują w cyberprzestrzeni. OWASP prowadzi szereg inicjatyw mających na celu edukację i świadomość w zakresie bezpieczeństwa aplikacji internetowych. Organizuje konferencje, warsztaty i szkolenia, które umożliwiają specjalistom ds. bezpieczeństwa oraz deweloperom zdobycie wiedzy i umiejętności potrzebnych do tworzenia bezpiecznych aplikacji. Dzięki tej inicjatywie twórcy oprogramowania mają dostęp do szerokiej wiedzy, aktualnych informacji oraz narzędzi, które umożliwiają im zapewnienie jak największego poziomu bezpieczeństwa aplikacji.

2.3 Analiza wybranych zagrożeń

Organizacja OWASP w swojej liście OWASP Top 10 2021, prezentuje zbiór dziesięciu najważniejszych zagrożeń bezpieczeństwa dotyczących aplikacji internetowych. Dodatkowo w najnowszej wersji lista została uaktualniona o dodatkowe trzy kategorie, które na rysunku 2.1 zaznaczone są kolorem zielonym.

Rys. 2.1 Lista OWASP Top 10 [opracowanie własne na podstawie zaktualizowanej listy OWASP Top 10 - 2021]

W kontekście zagrożeń aplikacji formularzowych, szczególną uwagę należy zwrócić na podatności sklasyfikowane jako:

- wstrzykiwanie (Injection),
- błędy w projekcie (Insecure Design),
- fałszywe żądania po stronie serwera (Server-Side Request Forgery).

Do kategorii Injection możemy zaliczyć ataki, takie jak wstrzykiwanie SQL [4], które polegają na wstrzyknięciu złośliwego kodu do zapytań lub poleceń. Występują wtedy, gdy dane wysyłane są do interpretera jako część polecenia, bądź zapytania. Wstrzyknięcie danych jest możliwe, kiedy w aplikacji nie ma zaimplementowanego odpowiedniego mechanizmu walidacji danych wejściowych. Skutkiem tego rodzaju ataku może być, np.: dostęp w trybie odczytu lub zapisu do całej bazy danych, ominięcie mechanizmu uwierzytelnienia, czy nawet odczyt wybranych plików w systemie. Kolejną kategorią sklasyfikowaną na liście OWASP jest podatność Cross-site Scripting (XSS) [5], który polega na wstrzyknięciu złośliwego kodu Java Script za pośrednictwem pola tekstowego lub parametru URL.

Wyróżniamy kilka rodzaj ataków XSS:

- Reflected,
- Stored,
- DOM-Based.

Reflected XSS występuje w przypadku, kiedy złośliwy kod JavaScript umieszczony jest w adresie URL. Ofiara klikając w link nieświadomie przekazuje złośliwe zapytanie do serwera aplikacji, następnie serwer zwraca złośliwy kod JavaScript przeglądarki i zostaje on wykonywany po stronie użytkownika. Ten rodzaj XSS jest niezwykle niebezpieczny, ponieważ może spowodować przejęcie kontroli m.in nad sesją użytkownika, wykradzenie haseł lub manipulowanie zawartością wyświetlaną po stronie ofiary.

Drugim rodzajem ataku Cross-site Scripting (XSS) jest *Stored XSS* [6], w porównaniu do Reflected XSS, w tym przypadku złośliwy skrypt umieszczany jest bezpośrednio na stronie lub w bazie danych. Kod wykonuje się za każdym razem, kiedy użytkownik odwiedza stronę, na której został wstrzyknięty złośliwy kod JavaScript.

Trzecim rodzajem XSS jest *DOM-Based XSS* [7], w tym przypadku payload wykonywany jest w wyniku modyfikacji DOM (Document Object Model) przeglądarki. Złośliwy kod JavaScript może zostać wstrzyknięty do adresu URL lub danych wejściowych. W tym przypadku odpowiedź serwera nie ulega zmianie, natomiast kod po stronie ofiary wykonywany jest zgodnie z zamierzeniami atakującego. Ten rodzaj ataku może skutkować wywołaniem niechcianych akcji oraz do manipulacji zawartością strony.

Kolejnym zagrożeniem należącym do kategorii “wstrzyknięcia” jest podatność aplikacji formularzowych na atak HPP [8]. Z tym rodzajem “wstrzyknięcia” można spotkać się, kiedy dane wejściowe w formularzach nie są czyszczone w odpowiedni sposób lub sprawdzane pod kątem poprawności. Atak polega na modyfikacji parametru HTTP, który może wpłynąć na zakłócenia w działaniu aplikacji, zmiany w jej logice, a nawet uzyskanie dostępu do zasobów, do których atakujący nie ma uprawnień.

Insecure Design [9] (niebezpieczny projekt) to kategoria, która po najnowszej aktualizacji w 2021, zajęła czwarte miejsce na liście OWASP Top 10. Podatność odnosi się do błędów, które powstały w fazie projektowania oprogramowania i w konsekwencji wprowadzania niewłaściwych metod zabezpieczeń. Niepoprawne zabezpieczenie aplikacji wiąże się z szeregiem podatności i ataków, na które aplikacja może być narażona. Brak podstawowych mechanizmów zabezpieczających, formularze takich jak: walidacja danych, mechanizmy kontroli dostępu, mechanizmy ochronne służące zabezpieczaniem przed atakami botów, stosowanie odpowiednich metod przesyłania

danych, a nawet niepoprawny interfejs użytkownika mogą doprowadzić do wielu problemów bezpieczeństwa aplikacji oraz jej użytkowników.

Kolejnym typem podatności, która jest istotna z perspektywy zabezpieczania aplikacji formularzowych jest SSRF [10]. Polega on na tym, że osoba atakująca wysyła żądanie HTTP z serwera aplikacji do innych zewnętrznych systemów, co w konsekwencji może umożliwić atakującemu uzyskanie dostępu do usług wewnętrznych aplikacji, wrażliwych danych oraz wykonanie nieautoryzowanych działań. Wysyłanie żądań do zewnętrznych źródeł możliwe jest w sytuacji, kiedy aplikacja nie jest właściwie zabezpieczona. W sytuacji, kiedy atakujący modyfikuje dane wejściowe i wysyła żądanie do serwera aplikacji, serwer przetwarza te dane i wykonuje żądanie HTTP do zewnętrznego lub wewnętrznego zasobu zgodnie z danymi dostarczonymi przez atakującego. Serwer wykonuje żądanie, a w konsekwencji może doprowadzić to do ominięcia mechanizmu zabezpieczeń systemu i uzyskanie dostępu do poufnych danych.

2.4 Metody zabezpieczania formularzy w aplikacjach internetowych

W celu zapewnienia właściwego poziomu bezpieczeństwa w aplikacjach formularzowych twórcy oprogramowania powinni korzystać z odpowiednich metod zabezpieczania, oferowanych przez nowoczesne technologie. Współcześnie wiele bibliotek zawiera wbudowaną funkcje sanityzacji danych, która automatycznie oczyszcza każdy dynamiczny tekst wprowadzony w aplikacji, w celu usunięcia potencjalnie niebezpiecznych elementów, takich jak niebezpieczne skrypty JavaScript

Wraz z rozwojem przeglądarek internetowych, ich producenci zaczęli implementować różnego rodzaju mechanizmy bezpieczeństwa, np.: Content Security Policy (CSP) [11], który umożliwia administratorom witryn precyzyjne określenie, jakie zasoby i z jakich domen, mogą być ładowane na ich stronie. Dodatkowo oprócz ograniczania konkretnych domen, mechanizm umożliwia ograniczenie wyświetlania strony do konkretnych protokołów, stanowi to istotny element w kwestii zabezpieczania witryny przed atakami, takimi jak: XSS. Współczesne środowisko programistyczne oferuje szereg wbudowanych metod służącym do zabezpieczania aplikacji, które stanowią kluczowy element do zapewnienia bezpieczeństwa jej użytkownikom oraz całego oprogramowania. W poniższych podrozdziałach zaprezentowano metody oraz techniki zabezpieczania danych wejściowych wprowadzanych przy pomocy formularzy. Dodatkowo przedstawione zostaną sposoby zabezpieczania baz danych, które będą zapobiegać nieautoryzowanym dostępom do aplikacji.

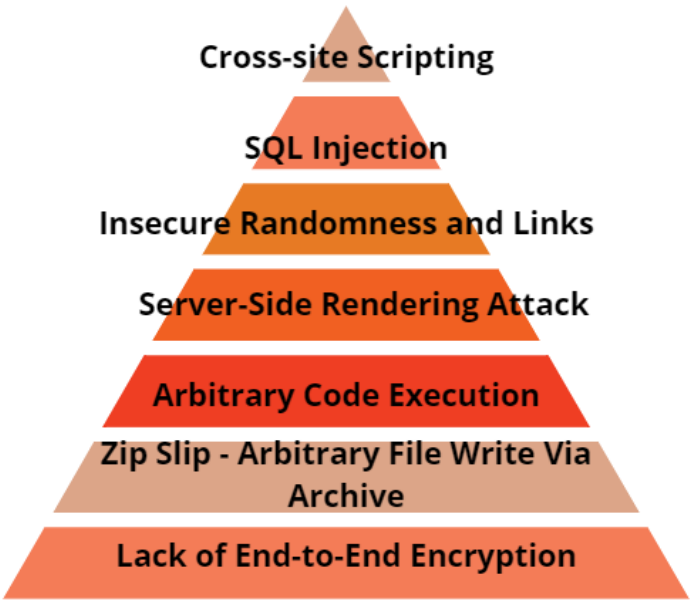
2.4.1 Nowoczesne biblioteki frontendowe

React [12] to obecnie jedna z najpopularniejszych bibliotek JavaScript [13], która umożliwia tworzenie interaktywnych i złożonych interfejsów użytkownika, obsługę danych wejściowych oraz manipulację nimi. Została stworzona przez firmę Meta i bazuje na modelu DOM, który służy do zarządzania całą strukturą i zawartością stron. Struktura Reacta oparta jest na komponentach, które dzielą interfejs na niezależne części, co pozwala na łatwiejsze utrzymanie aplikacji oraz nawigację po niej. Kluczową koncepcją jest możliwość zarządzania stanem komponentów, który wpływa na skalowalność oraz lepszą wydajność aplikacji. Dodatkowo charakteryzuje się wysoką kompatybilnością z innymi językami oraz z otwartymi kodami źródłowymi, co pozytywnie wpływa na optymalizację w kontekście wprowadzenia wyższego poziomu bezpieczeństwa, natomiast zwiększa to ryzyko podatności na ataki. Biblioteka React posiada wiele wbudowanych mechanizmów, które wpływają na zwiększenie poziomu bezpieczeństwa użytkowników oraz samej aplikacji. Jednym z popularnych mechanizmów Reacta jest automatyczne escapowanie wszystkich wartości wstawianych do JSX [15]. W sytuacji, kiedy atakujący chce wstrzyknąć złośliwy kod JavaScript do pola wejściowego w aplikacji, React automatycznie konwertuje potencjalnie, niebezpieczne znaki na ich bezpieczne odpowiedniki w encji HTML. Na przykład: znaki “<”,”>” są odpowiednio zamieniane na “&lt” i “&gt”. Mechanizm ten automatycznie przetwarza złośliwy kod jako wartość tekstową, a tym samym uniemożliwia generowanie złośliwego skryptu i chroni przed potencjalnym atakiem. Oprócz mechanizmu sanityzacji, biblioteka w kontekście ochrony oferuje wiele innych metod, które zabezpieczają aplikację. Kolejną biblioteką, która znalazła swoje zastosowanie w zapobiegania atakom typu XSS jest DOMPurify [16]. Narzędzie to oczyszcza dane wejściowe z niebezpiecznych i niepożądanych treści HTML, zapobiegając przedostaniu się szkodliwego kodu do aplikacji. W sytuacji, kiedy atakujący chce przesłać niebezpieczny skrypt JavaScript do pola wejściowego w aplikacji, narzędzie automatycznie oczyści go ze szkodliwych elementów i uniemożliwi potencjalny atak XSS. Kolejną biblioteką, która przyczynia się do zwiększenia poziomu bezpieczeństwa w aplikacjach React jest React-Router-Dom [17]. Odpowiada, między innymi za: regulację uprawnień użytkowników do konkretnych zasobów, dzięki kontrolowaniu dostępu i możliwością wprowadzania prywatnych ścieżek, do których dostęp mają tylko uprawnieni użytkownicy. Biblioteka odpowiada, również za zarządzanie historią nawigacji i bezpieczne przekierowanie na odpowiednią stronę, co jest niezwykle istotne podczas procesu logowania i rejestracji. React-Router umożliwia również implementacje uwierzytelniania dwuskładnikowego, które jest

szczególnie ważne w kontekście zapewnienia odpowiedniego poziomu bezpieczeństwa użytkownika oraz jego danych.

W aplikacjach typu SPA istotną rolę pełni biblioteka React-Helmet-Async [18], która umożliwia zarządzanie ważnymi elementami dokumentu HTML, w sekcji <head>. Jeśli nagłówek <head> będzie znajdował się bezpośrednio w komponencie React uniemożliwi to próbę jego modyfikacji. Koncepcja ta jest szczególnie ważna w aplikacjach jednostronicowych, gdzie zmiany nawigacji lub kontekstu nie powodują pełnego przeładowania strony. Biblioteka umożliwia bezpieczne zarządzanie zawartością <head> dzięki automatycznemu oczyszczaniu znaków, które wpływa na minimalizacje wystąpienia ataku XSS. Dodatkową zaletą, w kwestii zwiększenia bezpieczeństwa aplikacji jest hermetyzacja danych, która polega na tym, że każda sesja użytkownika jest traktowana jako oddzielna jednostka z własnym zakresem danych. Zabezpiecza to dane użytkownika przed możliwością wycieku poufnych informacji.

Otwarto źródłowy charakter Reacta zwiększa ryzyko podatności aplikacji na wszelkiego rodzaju luki bezpieczeństwa. Mimo, iż biblioteka posiada wiele wbudowanych narzędzi podnoszących ich poziom, programiści powinni stale aktualizować i ulepszać wiedzę dotyczącą najlepszych praktyk w zabezpieczaniu. Główne podatności Reacta zostały przedstawione na rysunku 2.2. Warto zwrócić uwagę na podatność Reacta na dwa najbardziej popularne ataki: Cross-Site Scripting oraz SQL Injection.



Rys. 2.2. Potencjalne luki bezpieczeństwa w React

[opracowanie własne, na podstawie: https://radixweb.com/blog/reactjs-security-vulnerabilities-and-solutions#why , *Dostęp: 4.05.2024]*

2.4.2 Sposoby walidacji danych wejściowych

Odpowiednia walidacja danych wejściowych w aplikacjach formularzowych jest kluczowym elementem zabezpieczania przed różnymi rodzajami ataków, szczególnie przed wstrzykiwaniem kodu SQL, XSS oraz SSRF. Właściwa implementacja walidacji chroni nie tylko aplikację, ale również jej użytkowników przed potencjalnym zagrożeniem oraz utratą wrażliwych danych. Wiele bibliotek oraz języków posiada wbudowane funkcje walidujące. Idealnym przykładem jest HTML5 – hipertekstowy język znaczników, który posiada wiele atrybutów służących prostej, ale skutecznej walidacji. Nie wymaga od programisty korzystania z JavaScript lub innych technologii, które mają służyć do walidacji danych wprowadzanych przez użytkownika. Atrybut "type" w HTML jest jednym z najpopularniejszych, ale również najbardziej podstawowym narzędziem służącym do definiowania typu danych, które użytkownik może wprowadzić w polu formularza. Dzięki temu atrybutowi przeglądarka może odpowiednio reagować na wprowadzone dane. Na rysunku 2.3 został przedstawiony kod HTML, który tworzy podstawowy formularz, służący do zbierania informacji za pomocą pól <input>. Wszystkie wpisane przez użytkownika dane są walidowane za poprzez odpowiednie typy atrybutów.

Przykładowo:

- "text" pozwala na wprowadzenie dowolnego tekstu,
- "email" przeznaczony do wprowadzania adresów e-mail, w tym przypadku przeglądarka sprawdza, czy tekst jest w formacie zgodnym z adresem e-mail,
- "date" w przeglądarka wymusza od użytkownika wybór daty w określonym formacie,
- "number" pole pozwala na wprowadzenie liczby, w tym przypadku można określić minimalną "min" oraz maksymalną "max" wartość,
- "password" w tym przypadku pole, do którego wprowadzone są znaki jest maskowane przez przeglądarkę.

```html
<h1>Formularz z różnymi typami inputów</h1>
<form>
    <label for="text">Tekst:</label>
    <input type="text" id="text" name="text"><br><br>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email"><br><br>

    <label for="date">Data:</label>
    <input type="date" id="date" name="date"><br><br>

    <label for="number">Numer:</label>
    <input type="number" id="number" name="number" min="1" max="100"><br><br>

    <label for="password">Hasło:</label>
    <input type="password" id="password" name="password"><br><br>

    <input type="submit" value="Wyślij">
    <input type="reset" value="Wyczyść">
</form>
```

Rys. 2.3. Przykłady podstawowych typów atrybutów w HTML [opracowanie własne]

Oprócz powyższych typów wykorzystywane są również inne, które szczególną rolę odgrywają w kwestii, np.: tworzenia bezpiecznych haseł przez użytkowników. Jednym z nich jest atrybut “pattern”, pozwalający określić konkretne wyrażenie regularne, którego wartość wpisana w polu powinna spełniać określone kryteria. Na rysunku 2.4 została przedstawiona walidacja hasła za pomocą atrybutu “pattern”, które musi: zawierać przynajmniej jedną dużą literę, jeden znak specjalny, jedną cyfrę oraz mieć co najmniej 10 znaków. Innym atrybutem, który służy do walidacji formularzy jest

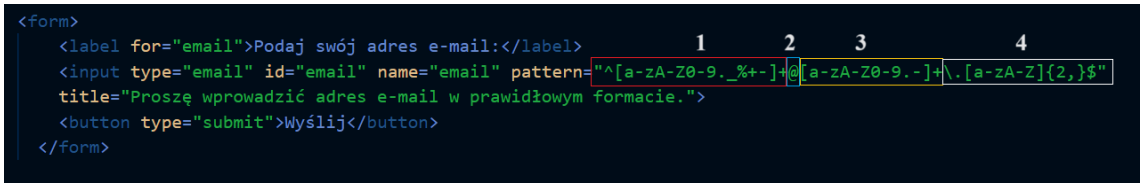
“required”, wymuszający na użytkowniku podanie lub wybranie określonej wartości w polu. Przeglądarka uniemożliwi wysłanie formularza w sytuacji, kiedy pole zostanie nieuzupełnione. Mechanizm ten zabezpiecza przed wysłaniem niekompletnych danych, a tym samym wpływa na integralność systemu oraz stanowi pewnego rodzaju zabezpieczenie przed atakami botów ze względu na utrudnienie w automatycznym przesyłaniu formularza.

```html
<form>
    <label for="password">Hasło:</label>
    <input type="password" id="password" name="password"
      pattern="^(?=.*[A-Z])(?=.*\d)(?=.*\W).{10,}$"
      title="Hasło musi zawierać przynajmniej jedną dużą literę, jedną cyfrę, jeden znak specjalny i mieć co najmniej 10 znaków.">
    <input type="submit">
</form>
```

Rys. 2.4. Walidacja hasła przy pomocy wyrażenia regularnego [opracowanie własne]

Szczególną uwagę, należy zwrócić na wyrażenia regularne - Regexp, które stanowią potężne narzędzie nie tylko w kwestii walidacji danych wejściowych, ale również

znalazły swoje zastosowanie jako wzorzec służący do wyszukania określonego tekstu i manipulację nim. Każde wyrażenie składa się z konkretnych wzorców, które w analizowanym tekście reprezentują odpowiednie sekwencje znaków. Na rysunku 2.5 została przestawiona walidacja adresu e-mail za pomocą odpowiedniego wyrażenia regularnego.



Rys. 2.5. Walidacja adresu e-mail za pomocą Regexp [opracowanie własne]

Wprowadzony przez użytkownika adres e-mail, powinien odpowiadać wzorcowi określonemu w atrybucie “pattern”:

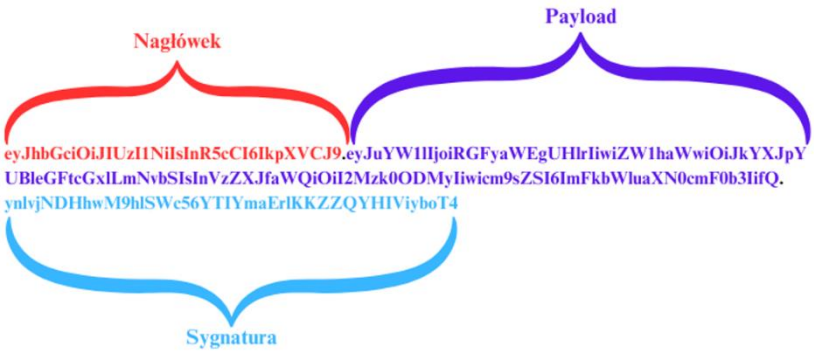
1. Początkowa cześć adresu może zawierać litery, cyfry oraz znaki specjalne,
2. Adres musi zawierać symbol “@”,
3. Pierwsza część domenowa adresu może składać się z cyfr, kropek, myślników oraz liter,
4. Druga część domenowa adresu musi zawierać przynajmniej dwie litery i być oddzielona od części pierwszej znakiem kropki.

Walidacja danych po stronie frontendu jest istotnym elementem, który zabezpiecza aplikację. W celu zwiększenia poziomu bezpieczeństwa dane wprowadzane do formularzy powinny być również zabezpieczane po stronie backendu, znacznie wpłynie to na zwiększenie integralności danych oraz podwyższy poziom bezpieczeństwa w całej aplikacji. Podwójna ochrona optymalizuje obciążenie serwera, ponieważ niekompletne lub nieprawidłowe dane odrzucane są już na poziomie przeglądarki, a do serwera wysłane zostają tylko te, które poprawbie przeszły proces walidacji. Właściwa walidacja po stronie backendu weryfikuje dane, które przesłane zostały do serwera. Sprawdza ich poprawność oraz to, czy nie zostały zmodyfikowane. Dwupoziomowa ochrona zabezpiecza aplikację przed różnego rodzaju atakami, a także poprawia stabilność i niezawodność systemu.

2.4.3 Zarządzanie sesją użytkownika i tokeny

Odpowiednie zarządzanie sesją użytkownika stanowi ważny element w kontekście bezpieczeństwa całej aplikacji oraz jej użytkowników. Zarządzanie sesjami obejmuje wiele technik i mechanizmów, które wpływają na zapewnienie bezpieczeństwa oprogramowania. Do popularnych rozwiązań można zaliczyć: JSON Web Tokens, polityki cookies oraz zarządzanie nagłówkami HTTP. Standard JSON Web Tokens (JWT) służy do wymiany informacji między klientem, a serwerem, w kontekście aplikacji internetowych i odgrywa kluczową role w uwierzytelnianiu oraz autoryzacji użytkownika. Stosowanie JWT daje gwarancję autentyczności danych przesyłanych do serwera aplikacji przez uwierzytelnionego użytkownika. Struktura tokenu składa się z trzech części: nagłówka, zawartości (payload) oraz sygnatury. Każda z tych części przechowuje inne informacje: nagłówek zawiera informacje o typie tokenu oraz algorytmie, który używany jest do jego zabezpieczenia. Wybór algorytmu jest ważny w kontekście poziomu bezpieczeństwa, ponieważ silne algorytmy kryptograficzne uniemożliwiają złamanie zaszyfrowanego tokenu. Payload zawiera między innymi

szczegółowe informacje dotyczące identyfikatora użytkownika, który jest właścicielem tokenu, czas wygaśnięcia tokenu oraz uprawnienia użytkownika, które określają poziom dostępu do zasobów. Część zawierająca sygnaturę służy do weryfikacji tokenu, tworzona jest przez zaszyfrowanie połączonego nagłówka, payloadu oraz tajnego klucza. Sygnatura to cześć, która bezpośrednio odpowiada za weryfikację tokenu przez serwer, ponieważ jeśli, któraś z zakodowanych części zostanie zmodyfikowana spowoduje to utratę jej ważności. JWT powinny być przechowywane po stronie klienta, w celu zabezpieczenia przed atakami XSS. Na rysunku 2.6 został zaprezentowany przykładowy token, który w każdej części ma zakodowane konkretne informacje.



Rys. 2.6. Przykładowy JSON Wen Token [opracowanie własne]

Na rysunku 2.7 zostały zaprezentowane przykładowe zdekodowane informacje, które szczegółowo opisują informacje w nim zawarte.

Rys. 2.7. Zdekodowane części JSON Web Token [opracowanie własne]

Powyższe zdekodowane, w poszczególnych częściach określają konkretne informacje:

- nagłówek: algorytm kryptograficzny HMAC z funkcja haszującą SHA-256, typ tokenu: JSON Web Token,
- payload: zawiera informacje dotyczące użytkownika tj. Imię i nazwisko, adres e-mail, unikalny identyfikator, który posiada w systemie oraz rolę jaka jest mu przyznana,
- sygnatura: zawiera konkatenację zakodowanej części nagłówka i payload'u w Base64Url oraz dodatkowy zakodowany tajny klucz, który jest stringiem o wartości "mySecretKey123!".

Druga ważną techniką jest polityka ciasteczek, która odgrywa kluczową rolę w zarządzaniu prywatnością użytkowników i zapewnieniu bezpieczeństwa danych. Cookies zawierają informacje dotyczące sesji użytkownika, jego preferencji i inne ustawienia. W celu zwiększenia bezpieczeństwa użytkowników oraz ochrony przed różnymi atakami, stosuje się konkretne atrybuty. Jednym z podstawowych jest flaga ”Secure”, który gwarantuje, że pliki mogą zostać wysłane tylko i wyłącznie, kiedy dana strona korzysta z bezpiecznego protokołu HTTPS, zapewniającego szyfrowanie danych. Atrybut chroni przed nieuprawnionym dostępem do informacji, które przechowywane są w plikach przesyłanych pomiędzy przeglądarką, a serwerem. Niemniej jednak stanowi on jedynie podstawowe zabezpieczenie i nie chroni użytkownika przed XSS i SSRF oraz nie zapewnia pewnej integralności przesyłanych danych. W celu ochrony przed powyższymi atakami, podczas implementacji zabezpieczeń należy skorzystać z dodatkowego atrybutów “HttpOnly”, “SameSite” oraz mechanizmu takiego jak tokeny anty-CSRF. “HttpOnly” uniemożliwia dostęp do cookies przez złośliwe skrypty, które znajdują się po stronie klienta, co stanowi ważne zabezpieczenie w kontekście ochrony przed atakami XSS. Ograniczenie dostępu z poziomu skryptów klienta umożliwia ochronę danych użytkownika, które są w nich przechowywane. Mimo, że atrybut “HttpOnly” stanowi ważne zabezpieczenie w kontekście ochrony przed atakami XSS, to nie jest on odpowiednią metodą służącą do ochrony przed SSRF. W celu zabezpieczenia przed tego rodzaju atakiem należy zaimplementować dodatkowy atrybut “SameSite”, odpowiadający za obsługę żądań między różnymi domenami. Przyjmuje on konkretną wartość, w zależności od ograniczeń, które chcemy, żeby aplikacja posiadała. Przykładowo wartość “Strict” uniemożliwia wysyłanie cookie do zapytań pochodzących z innych domen, stanowi to najbezpieczniejszą opcję w przypadku ataków SSRF, ponieważ w sytuacji, kiedy użytkownik kliknie w złośliwy link, który prowadzi go na inną stronę, cookies nie zostaną przesłane. Kolejną wartością jest “Lax”, to zwykle domyślna i najczęściej spotykana konfiguracja atrybutu “SameSite”, pozwala na wysyłanie standardowych żądań typu GET, natomiast ogranicza żądania POST, ponieważ nie będą one zawierały cookies. Atrybuty “SameSite” oraz “HttpOnly” w połączeniu ze sobą stanowią istotne zabezpieczenie zarówno przed atakami XSS oraz SSRF. Implementacja tych mechanizmów pomaga w utrzymaniu integralności sesji oraz zabezpiecza aplikację przed nieautoryzowanym dostępem, co jest kluczowe w kwestii bezpieczeństwa aplikacji internetowych. Na rysunku 2.8 zaprezentowano przykładową implementację powyższych zabezpieczeń dla ciasteczek wraz z ograniczeniem czasu, przez który są ważne. Właściwości zaimplementowane są na poziomie serwera aplikacji,

przy wykorzystaniu freamworka Express.js, który działa w środowisku wykonawczym Node.js.

```
app.get('/notes', async (req, res) => {
  try {
    res.cookie('userSession', 'randomValueOfCookie', {
      httpOnly: true,
      secure: true,
      sameSite: 'strict',
      maxAge: 3600000
    });

    const result = await db.query('SELECT * FROM note');
    res.json(result.rows);
  } catch (err) {
    console.error(err);
    res.status(500).send('Internal Server Error');
  }
});
```

Rys. 2.8. Atrybuty zabezpieczające: HttpOnly, Secure, SameSite dla endpiitu GET [opracowanie własne]

Wszelkie informacje, które przechowywane są po stronie klienta mogą znajdować się nie tylko w cookies, ale również w “localStorage” lub “sessionStorage”. Dane przechowywane w localStorage są przez czas nieokreślony i pozostają dostępne po zamknięciu i ponownym otwarciu przeglądarki. Dostępne są dla całej domeny, więc nie są odpowiednie do przechowywania wrażliwych danych. Stosowane są do przechowywania informacji o stanie lub preferencjach użytkowników. “SessionStorage” przechowuje dane tylko przez okres trwania sesji przeglądarki. Po jej zamknięciu dane są usuwane, a dostępność do nich jest jedynie w obrębie karty, a nie całej domeny. Ten sposób magazynowania danych jest nieco bezpieczniejszy, ponieważ umożliwia kontrolowane usuwanie informacji. Jest również użyteczny w kontekście zapisywania danych związanych jedynie z pojedynczą sesją użytkownika, na przykład w formularzach internetowych, w trakcie ich wypełniania.

Kolejny mechanizm zabezpieczający dotyczy bezpośrednio nagłówków HTTP, które związane są z komunikacją pomiędzy użytkownikami, a serwerami. W nich zawarte

są równie istotne informacje, dotyczące użytkownika oraz serwera. Nagłówkiem, który zabezpiecza przed atakiem XSS jest *Content Security Policy (CSP)* - Polityka Bezpieczeństwa zasobów, reguluje ona zasoby, które mogą być ładowane na stronie oraz źródła, z których mogą być wyświetlane. Przykładowe dyrektywy znajdują się w tabeli 2.1, każda z nich definiuje różne reguły pobierania zasobów.

Tabela 2.1. Przykładowe dyrektywy CSP [opracowanie własne]

Dyrektywa	Przykład	Znaczenie
`img-src`	`img-src *.example.com`	Dyrektywa umożliwia ładowanie obrazów jedynie z subdomen konkretnej domeny 'example.com'
`font-src`	`font-src 'self' https://fonts.googleapis.com`	Dyrektywa kontroluje, skąd mogą być ładowane czcionki na stronie (w przykładzie to samo źródło oraz strona z Google Fonts)
`style-src`	`style-src 'self' https://example.com`	Dyrektywa kontroluje skąd mogą być ładowane arkusze stylów, w przykładzie ograniczenie do tego samego źródła co domena lub konkretna 'example.com'
`script-src`	`script-src 'self' https://example.com`	Dyrektywa definiuje źródło skryptu JS ograniczone do konkretnego adresu URL

Polityka bezpieczeństwa treści, czyli CSP jest kluczowym narzędziem, które szczególną rolę odgrywa w zapewnieniu odpowiedniego poziomu bezpieczeństwa w aplikacjach internetowych. Często stosowana jest jedynie jako dodatkowa warstwa zabezpieczeń, niemniej jednak jest to niezwykle potężne narzędzie przynoszące wiele korzyści. Dyrektywy CSP pozwalają na szczegółową kontrolę nad treściami, które mogą być wyświetlane oraz wykonywane na stronie, w znacznym stopniu poprawia to bezpieczeństwo użytkowników oraz aplikacji. CSP jest również stosowane w celu zminimalizowania ryzyka ataków typu XSS. W dobie współczesnych aplikacji internetowych właściwa implementacja skutecznych mechanizmów do zarządzania danymi użytkowników i sesjami jest kluczowa w zapewnieniu odpowiedniego poziomu bezpieczeństwa oraz efektywnego przetwarzania żądań pomiędzy klientem, a serwerem. Odpowiednia implementacja mechanizmów zarządzających tokenami, polityką ciasteczek oraz nagłówkami HTTP może pozytywnie wpłynąć na bezpieczeństwo

aplikacji, dodatkowo zabezpieczenia powinny być stale aktualizowane i dostosowane do aktualnych zagrożeń.

2.4.4 Techniki zabezpieczania baz danych

Systemy bazodanowe są szczególnie ważną częścią aplikacji internetowych, która wymaga implementacji odpowiednich zabezpieczeń. W bazach danych aplikacji internetowych przechowywane są informacje, które dotyczą użytkowników, ale również samej aplikacji. Informacje gromadzone w bazach mogą zawierać między innymi: dane osobowe, logowania, informacje dotyczące praw dostępu do konkretnych zasobów systemowych, dane konfiguracyjne oraz inne poufne i wrażliwe dane. Kluczowym elementem w kwestii zabezpieczenia bazy danych jest właściwe budowanie zapytań, które chronią przed atakami wstrzykiwania złośliwego kodu SQL. Tego rodzaju ataki polegają na wstrzyknięciu złośliwego kodu SQL bezpośrednio do zapytania. Powoduje to modyfikację właściwego zapytania, w wyniku którego wykonywana jest niepożądana akcja. Skutki udanego ataku mogą być bardzo poważne, ponieważ mogą doprowadzić między innymi do nieautoryzowanemu dostępu do aplikacji, manipulacją oraz zniszczeniem danych. Podatność tego typu występuje w sytuacji, kiedy aplikacja pobiera dane z formularza bezpośrednio do zapytania bazodanowego. Sposobem ochrony na ten rodzaj ataków jest stosowanie *Object Relational Mapping (ORM)* – mapowania obiektowo-relacyjnego oraz parametryzacja zapytań. ORM jest techniką programowania, która polega odwzorowaniu obiektowej architektury systemu informatycznego na relacyjną bazę danych. W kontekście aplikacji internetowych i baz danych, technika ta pozwala na mapowanie danych przechowywanych w tabelach na obiekty w językach programowania. Przykładem może być biblioteka Sequelize [18], która jest nowoczesnym ORM stosowanym między innymi dla PostgreSQL, MySQL oraz MariaDB. Narzędzie to umożliwia deweloperom korzystanie z relacyjnych baz danych w bardzo intuicyjny i obiektowy sposób. Pełni również ważną rolę w kontekście zabezpieczania przed atakami wstrzyknięcia kodu SQL. Posiada on wbudowany mechanizm parametryzacji zapytań, który w sposób automatyczny waliduje dane od użytkownika, sprawdzając, czy nie zawierają one złośliwych fragmentów kodu. Dzięki temu, wprowadzone parametry są automatycznie obsługiwane i traktowane jako dane, a nie jako część zapytania SQL. Mechanizm ten polega na oddzieleniu danych wejściowych użytkownika od instrukcji SQL, co umożliwia wykrycie złośliwych poleceń przesyłanych do serwera jako część zapytania. Parametryzacja zapytań jest jedną z najbardziej skutecznych metod służących do zapobiegania wstrzyknięciu kodu SQL.

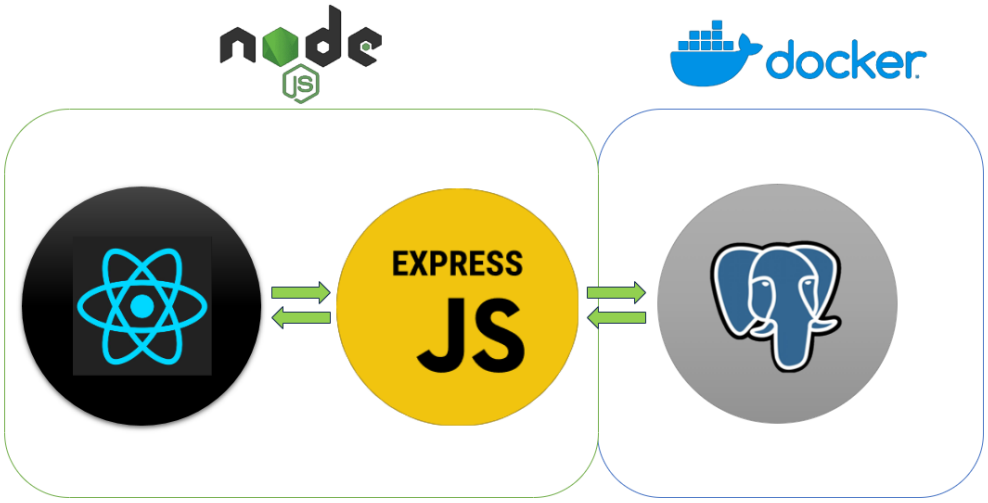
Kolejną techniką stosowaną w celu ochrony danych przechowywanych, w bazach aplikacji internetowych jest ich szyfrowanie. Kluczową kwestią tej metody jest implementacja szyfrowania haseł użytkowników, które przechowywane są w bazach danych. W sytuacji, kiedy nieautoryzowany użytkownik uzyska dostęp do bazy haseł, nie jest w stanie ich odczytać, ponieważ nie są one zapisane w formie jawnej. Stanowi to dodatkową metodę zabezpieczenia i minimalizuje ryzyko nieautoryzowanego dostępu do danych. Biblioteką używaną do haszowania haseł jest bcrypt, opiera się na szyfrze Blowfish i zapewnia podwójne bezpieczeństwo poprzez implementację mechanizmu solenia oraz wielokrotnego haszowania. Sól jest losowo generowanym ciągiem znaków, który dodawany jest do każdego hasła przed jego haszowaniem, natomiast mechanizm wielokrotnego haszowania zwiększa czas potrzebny do złamania hasła, co stanowi barierę dla ataków brute-force. Szkodliwe działania mogą doprowadzić do zmiany, usunięcia, zniszczenia, wykradnięcia wrażliwych danych, a nawet wykonywania dowolnych poleceń na serwerze bazy danych, co w konsekwencji może doprowadzić do naruszenia spójności całego systemu, dlatego tak ważna jest odpowiednia implementacja zabezpieczeń.

Część badawcza

3. Implementacja rozwiązania

3.1 Aplikacja ToDo List

W ramach pracy badawczej stworzono prostą aplikację formularzową ToDo List, która oferuje podstawowe funkcjonalności w zarządzaniu zadaniami. Obsługiwane są cztery podstawowe operacje CRUD – Create, Read, Update oraz Delete, które stanowią podłoże do przeprowadzenia testów penetracyjnych oraz implementacje zabezpieczeń przeciwko konkretnym atakom. Rozwiązanie posiada, również zaimplementowany mechanizm uwierzytelniania użytkownika. Do stworzenia aplikacji wykorzystano powszechnie stosowane rozwiązania technologiczne, takie jak: React, Express.js oraz PostgreSQL. Są to popularne narzędzia wykorzystywane podczas tworzenia aplikacji internetowych. React jest biblioteką JavaScript, która służy do budowania interaktywnych interfejsów użytkownika, w oparciu o niezależne komponenty. Backend aplikacji stworzono w oparciu o framework Express.js, który pozwala na obsługę żądań i integrację z bazami danych. System bazodanowy, który służył do przechowywania danych był PostgreSQL, to zaawansowane narzędzie umożliwiające tworzenie i zarządzanie relacyjnymi bazami. Środowisko bazodanowe zostało skonfigurowane i uruchomione na kontenerach Docker, który umożliwia wirtualizacje aplikacji i oparte jest na architekturze trzech niezależnych mikroserwisów. Wizualizacja architektury została przedstawiona na rysunku Rys. 3.1.



Rys. 3.1. Architektura aplikacji ToDo List [opracowanie własne]

Stworzone rozwiązanie było punktem wyjścia do przetestowania podatności prostej aplikacji formularzowej, w kontekście ataków takich jak: wstrzyknięcie kodu SQL, XSS, SSRF oraz HPP. Kolejnym etapem była implementacja odpowiednich zabezpieczeń i ponowne przetestowanie aplikacji. Wszystkie etapy zostały szczegółowo opisane w poniższych rozdziałach, wraz z wyszczególnieniem narzędzi i technologii, które zostały użyte do stworzenia i przetestowania oprogramowania [19].

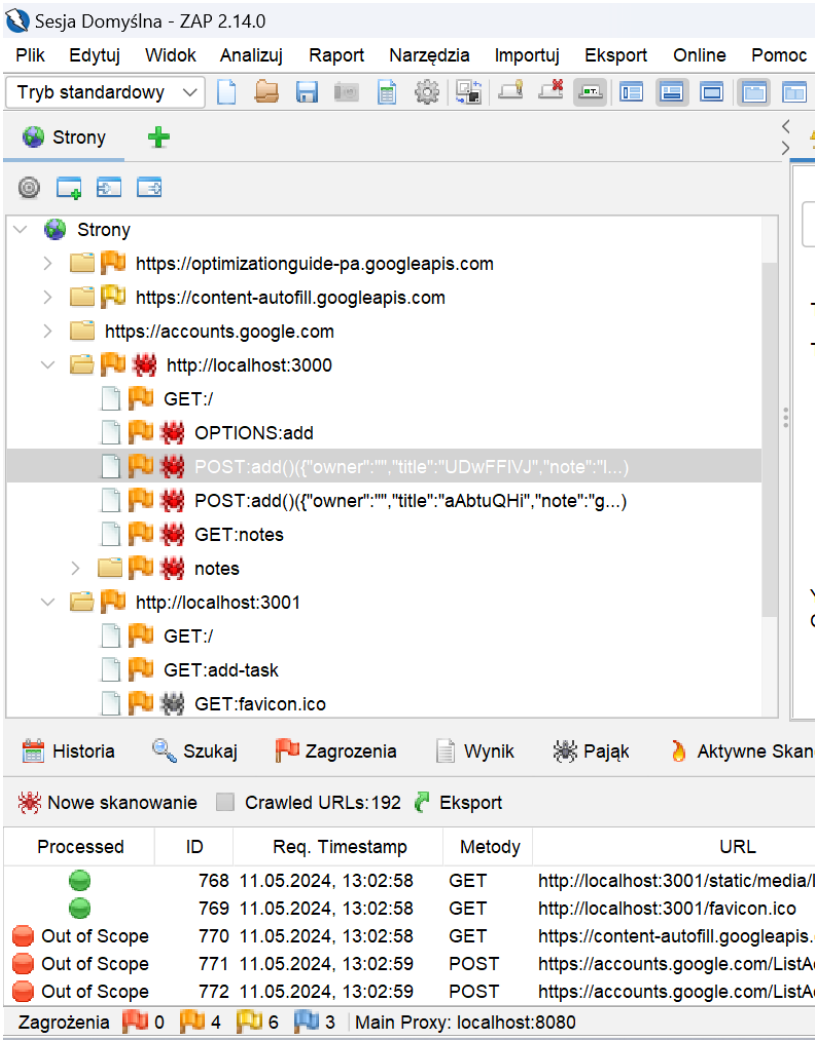
3.2 Opis wykorzystanych narzędzi do testowania aplikacji

W ramach testów penetracyjnych, które zostały przeprowadzone na potrzeby części badawczej niniejszej pracy, korzystano z dwóch narzędzi: ZAP w wersji 2.14 oraz aplikacji Postman. Używane są one do testowania bezpieczeństwa aplikacji webowych i zapewniają kompleksową analizę potencjalnych zagrożeń. Narzędzie ZAP jest rozwijane przez fundację OWASP i umożliwia automatyczne skanowanie aplikacji w poszukiwaniu luk bezpieczeństwa, które umożliwiają przeprowadzenia ataków, takich jak: wstrzyknięcie złośliwego kodu SQL, XSS oraz SSRF. Oprócz automatycznego testowania, narzędzie umożliwia testerom przeprowadzenie ich ręcznie, dzięki temu możliwe jest bardziej precyzyjne przeprowadzenie i dostosowanie ich do specyfiki aplikacji. Kolejną aplikacją, która posłużyła jako narzędzie do testowania to Postman, który umożliwiające tworzenie i wysłanie żądań do punktów końcowych API. Dodatkowym atutem jest testowanie uwierzytelniania użytkownika, ponieważ Postman pozwala na dodawanie tokenów do nagłówków żądań, co pozwala na dokładne testowanie mechanizmów uwierzytelniania i autoryzacji.

3.3 Opis implementacji zabezpieczeń przeciwko konkretnym atakom

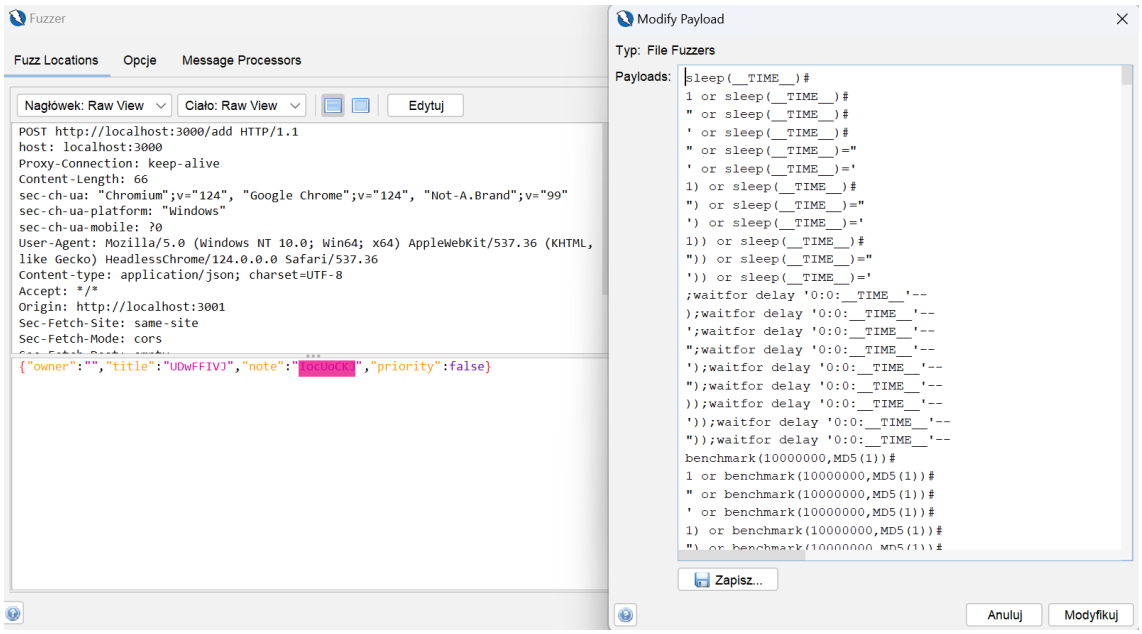
3.3.1 Wstrzyknięcie złośliwego kodu SQL

W wersji podstawowej aplikacji zapytania do bazy danych były tworzone poprzez bezpośrednio przekazywane wartości pochodzące z formularza do zapytania SQL. Tego typu podejście stanowi bezpośrednie zagrożenie dla bezpieczeństwa i umożliwia przeprowadzenie ataku wstrzyknięcia kodu SQL. W ramach pracy badawczej, z wykorzystaniem narzędzia ZAP przeskanowano aplikację, w celu wykrycia endpointów wykorzystywanych do komunikacji z bazą danych. Efekt działania zaprezentowano na rysunku 3.2.

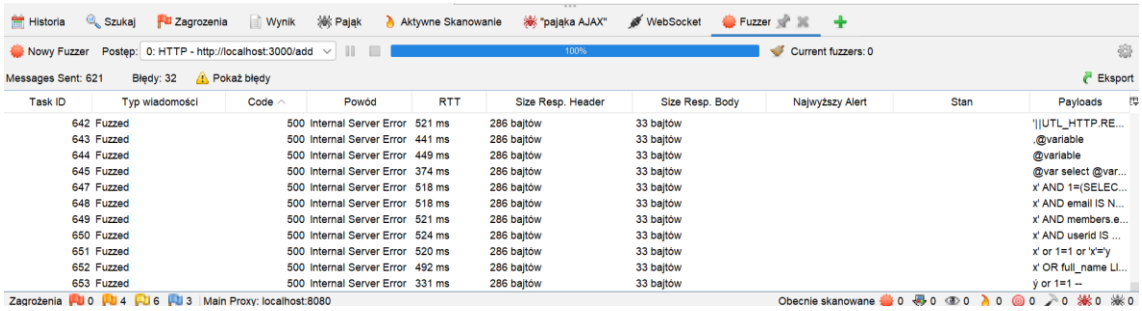


Rys. 3.2. Endpinty służące do komunikacji z bazą danych przeskanowanej aplikacji [opracowanie własne]

Przy pomocy wcześniej wspomnianego narzędzia przeprowadzono atak polegający na przesłaniu zmodyfikowanych parametrów, których celem jest wykonanie niepożądanych akcji w bazie danych. Na rysunku 3.3 zaprezentowano sposób przeprowadzenia ataku SQL Injection przy pomocy narzędzia ZAP, z wykorzystaniem parametru “note”, w ciele zapytania. Efektem przeprowadzonego ataku było usunięcie wszystkich rekordów, które znajdowały się w bazie danych.

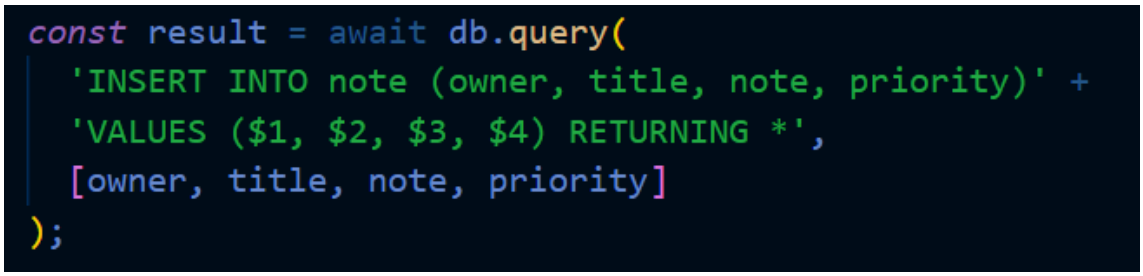


Rys. 3.3. Przygotowanie ataku SQL Injection w ZAP [opracowanie własne]



Rys. 3.4. Tabela z wynikami przeprowadzonych ataków SQL Injection [opracowanie własne]

W celu zabezpieczenia aplikacji przed tego typu atakami wprowadzono parametryzacje zapytań. Dzięki zastosowaniu tego mechanizmu dane wprowadzone przez użytkownika są walidowanie i przypadku wykrycia złośliwego kodu zapytanie SQL nie jest wykonywane. Na rysunku 3.5 został przedstawiony fragment kodu odpowiadający za parametryzację zapytania. Wprowadzone w nim wartości, które przekazywane są jako elementy tablicy “[owner, title, note, priority]”, są automatycznie mapowane z symbolami “($1, $2, $3, $4)”. Dzięki temu wartości wejściowe traktowane są jako dane, a nie jako kod SQL. Znaki specjalne, które zostają wprowadzone, traktowane są jako zwykłe znaki tekstowe, co uniemożliwia zmianę struktury całego zapytania.



Rys. 3.5. Parametryzacja zapytania - sposób ochrony przed SQL Injection [opracowanie własne]

Po wprowadzeniu modyfikacji kodu, ponownie przetestowano aplikację. Na podstawie uzyskanych wyników, można jednoznacznie stwierdzić, że żadne z ataków nie powiódł się. Parametryzacja zapewnia, że dane wejściowe są automatycznie oddzielone od kodu SQL, więc struktura zapytania pozostaje niezmienna bez względu na wartości, które użytkownik przekazuje w zapytaniu. Zatem można stwierdzić, że stosowanie bibliotek, które umożliwiają parametryzację zapytań w sposób wystarczający chronią aplikację przed atakami SQL Injection. Drugim sposobem na ochronę przed tego typu atakiem jest zastosowanie ORM, czyli narzędzia służącego do mapowania obiektowo-relacyjnego. Odzwierciedla obiekty bazodanowe na obiekty w aplikacji. Główna różnica między tymi dwoma podejściami jest taka, że w przypadku ORM zapytania są generowane w sposób automatyczny, dzięki czemu zmniejsza się ryzyko związane z niepoprawną budową zapytania. Niemniej jednak, oba te rozwiązani weryfikują przekazywane parametry, pod kątem złośliwego kodu. Jeśli zostanie wykryte jakieś zagrożenie, zapytanie nie zostanie wysłane do bazy oraz aplikacja zwróci błąd. W ramach pracy badawczej przetestowano również to rozwiązanie, okazało się ono równie skuteczne co parametryzacja zapytań. Na rysunkach 3.6 oraz 3.7 zaprezentowano użycie ORM, przy wykorzystaniu biblioteki “Sequelize”, która umożliwia połączenie się aplikacji napisanej w języku JavaScript z bazą PostgreSQL Wybór technologii zabezpieczeń w dużej mierze zależy od specyfiki projektu, niemniej jednak oba te rozwiązania zapewniają skuteczną ochronę przed atakami SQL Injection.

```
const Note = sequelize.define('Note', {
  owner: {
    type: DataTypes.INTEGER,
    allowNull: false
  },
  title: {
    type: DataTypes.STRING,
    allowNull: false
  },
  note: {
    type: DataTypes.STRING,
    allowNull: false
  },
  priority: {
    type: DataTypes.BOOLEAN,
    allowNull: true
  }
}, {
  tableName: 'note',
  timestamps: false,
  underscored: true
});
```

Rys. 3.6. Zdefiniowanie obiektu na obiekt aplikacji [opracowanie własne]

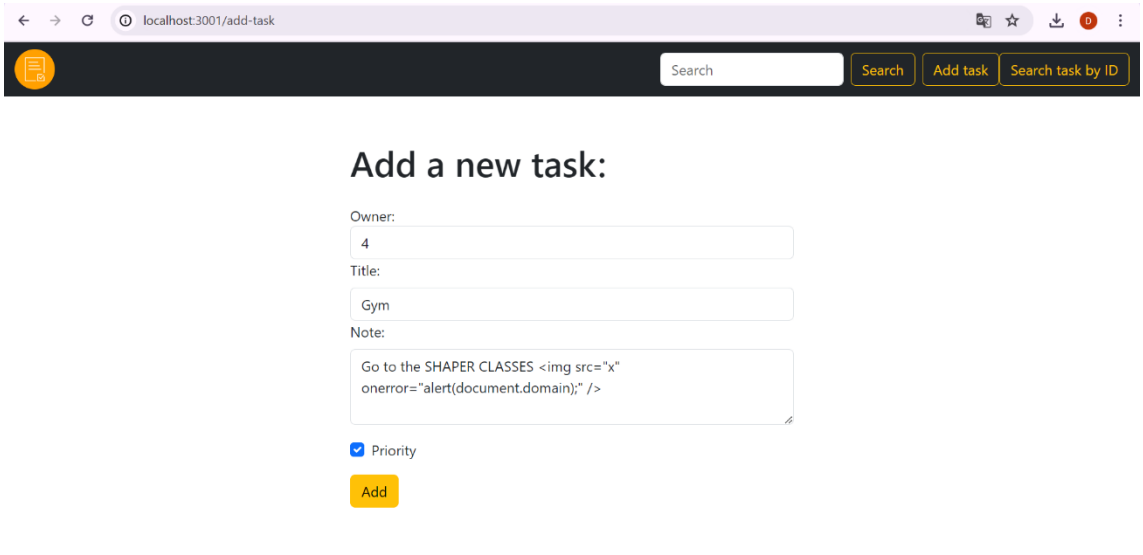
```
app.get('/notes/:id', async (req, res) => {
  const { id } = req.params;
  try {
    const result = await Note.findAll({
      where: {
        id: {
          [Op.eq]: id
        }
      }
    });
    if (result.length === 0) {
      res.status(404).json({ error: 'Note not found' });
    } else {
      res.json(result);
    }

  } catch (err) {
    console.error('Error executing query', err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

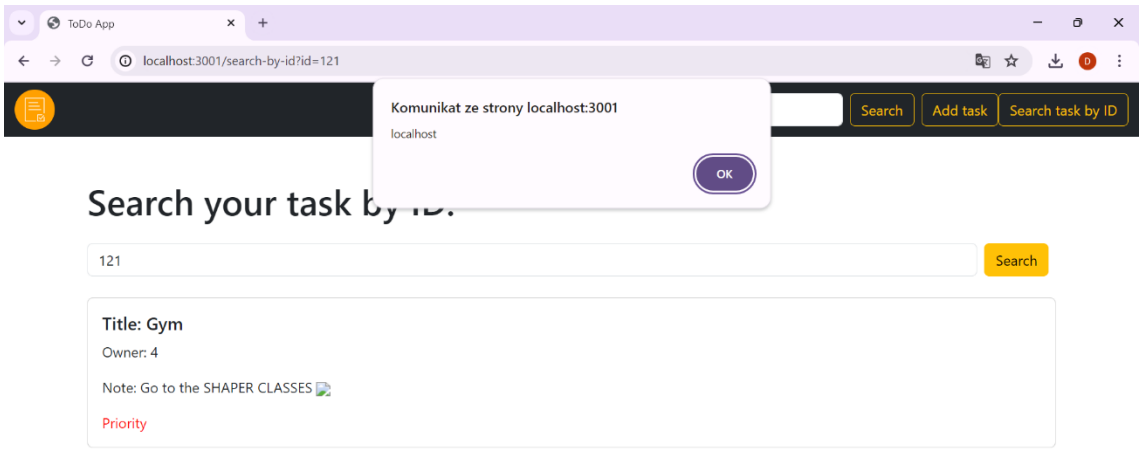
Rys. 3.7. ORM zastosowany w endpoincie GET [opracowanie własne]

3.3.2 XSS

W wersji podstawowej aplikacji wykorzystano atrybut “dangerouslySetInnerHTML” [20], który umożliwia wstawienie treści HTML pochodzących z zewnętrznych źródeł i ich dynamiczne generowanie na stronie. Tego typu podejście otwiera nowy wektor ataku na aplikację i umożliwia skuteczne przeprowadzenia ataku typu XSS. Brak walidacji zewnętrznego kodu HTML stanowi poważne zagrożenie dla bezpieczeństwa aplikacji, ponieważ może zostać w nim ukryty złośliwy skrypt. Wykorzystany atrybut “dangerouslySetInnerHTML”, umożliwienia wyświetlenie dowolnej treści w jednym z pól formularza. Podejście to jest funkcjonalne, ponieważ umożliwia zamieszczanie tekstów i linków pochodzących z, zewnętrznych źródeł. W ramach ataku przeprowadzonego podczas badań, wykorzystano znacznik HTML typu image, umożliwił on ukrycie złośliwego skryptu w kodzie obsługi błędu. Dzięki zastosowania tego podejścia skrypt zostanie wywołany w przeglądarce użytkownika, w momencie, kiedy źródło obrazu nie zostanie znalezione. W ramach testu wykorzystano przykładowy złośliwy kod, którego celem jest wyświetlanie komunikatu z nazwą domeny. Atak został zakończony sukcesem. Na rysunku 3.8 zaprezentowano dodawanie zadania wraz ze złośliwym kodem, natomiast rysunek 3.9 prezentuje efekt jego działania.



Rys. 3.8. Umieszczenie złośliwego kodu w polu służącym do dodawania zadania [opracowanie własme]



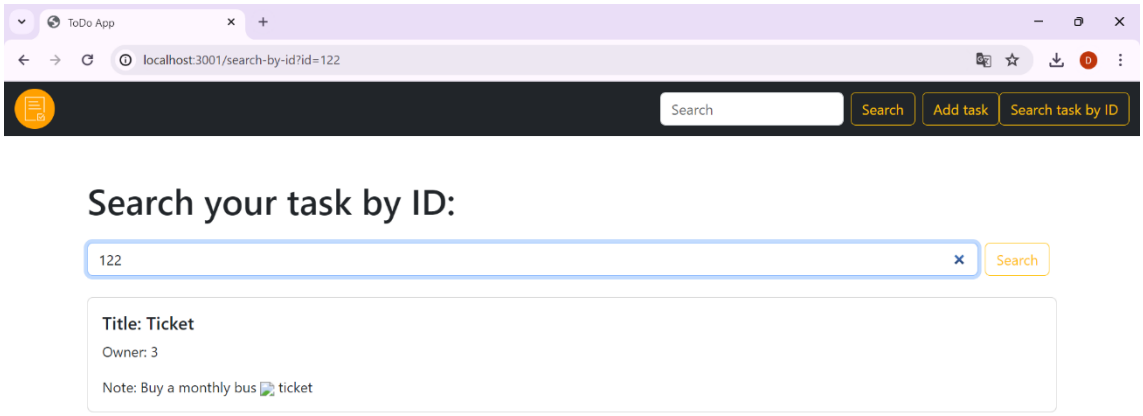
Rys. 3.9. Wynik działania złośliwego skryptu [opracowanie własne]

W celu zabezpieczenia aplikacji przed tego typu atakami, wprowadzono bibliotekę "DOMPurify", która automatycznie oczyszcza potencjalnie niebezpieczne elementy HTML wprowadzane przez użytkownika. Funkcja "sanitize" usuwa niebezpieczne znaczniki i atrybuty z treści HTML i pozostawia tylko te, które uznawane są za bezpieczne. Dzięki temu w sytuacji, kiedy w formularzu zostanie wprowadzony złośliwy kod, zostanie on automatycznie usunięty i nie zostanie wywołany w przeglądarce użytkownika. Funkcja "DOMPurify.sanitize" odpowiada za oczyszczanie danych pochodzących od użytkownika, zawartych w zmiennej "task.note". Kolejno dane te wstawiane są do DOM za pomocą atrybutu "dangerouslySetInnerHTML", który odpowiada za bezpośrednie wstawianie HTML do elementów React. Tego typu podejście stanowi ważne zabezpieczenie przed atakami typu XSS. Na rysunku 3.10 został zaprezentowany fragment kodu wraz z implementacją biblioteki "DOMPurify" oraz funkcją "sanitize" należącej do niej.

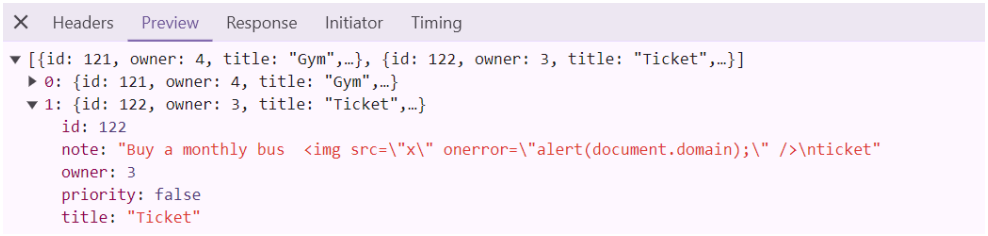
```
<div>
    {foundTask.map(task => (
        <div className="card mt-4" key={task.id}>
            <div className="card-body">
                <h5 className="card-title">Title: {task.title}</h5>
                <p className="card-text">Owner: {task.owner}</p>
                <p className="card-text" dangerouslySetInnerHTML={{__html: DOMPurify.sanitize(`Note: ${task.note}`)}} />
                {task.priority && (
                    <p className="card-text" style={{ color: 'red' }}> Priority {task.priority}</p>
                )}
            </div>
        </div>
    ))}
</div>
```

Rys. 3. 10. Zastosowanie biblioteki "DOMPurify" [opracowanie własne]

Po zaimplementowaniu biblioteki “DOMPurify” przeprowadzono atak ponownie, tym razem nie zakończył się on sukcesem. Na rysunku 3.11 przedstawiono proces wyszukiwania zadania ze złośliwym skryptem, podczas jego wyświetlania nie doszło do pojawienia się alertu. Natomiast na rysunku 3.12 został przedstawiony złośliwy skrypt zawarty w ciele zapytania HTTP. Rysunek 3.13, przedstawia oczyszczony kod przez bibliotekę z niebezpiecznych znaczników.



Rys. 3.11. Wyszukiwanie zadania po konkretnym numerze id [opracowanie własne]



Rys. 3.12. Oryginalne dane z zapytania HTTP [opracowanie własne]

Rys. 3.13. Oczyszczony znacznik HTML w przeglądarce [opracowanie własne]

Wykorzystanie narzędzie “DOMPurify” do oczyszczania danych wejściowych w znaczny sposób podnosi poziom bezpieczeństwa aplikacji pod względem podatności na ataki typu XSS. Dzięki tej bibliotece programiści mogą wykorzystywać atrybut “dangerouslySetInnerHTML” bez obawy, że w ten sposób zostanie stworzona luka w zabezpieczeniach, która może posłużyć jako wektor ataku.

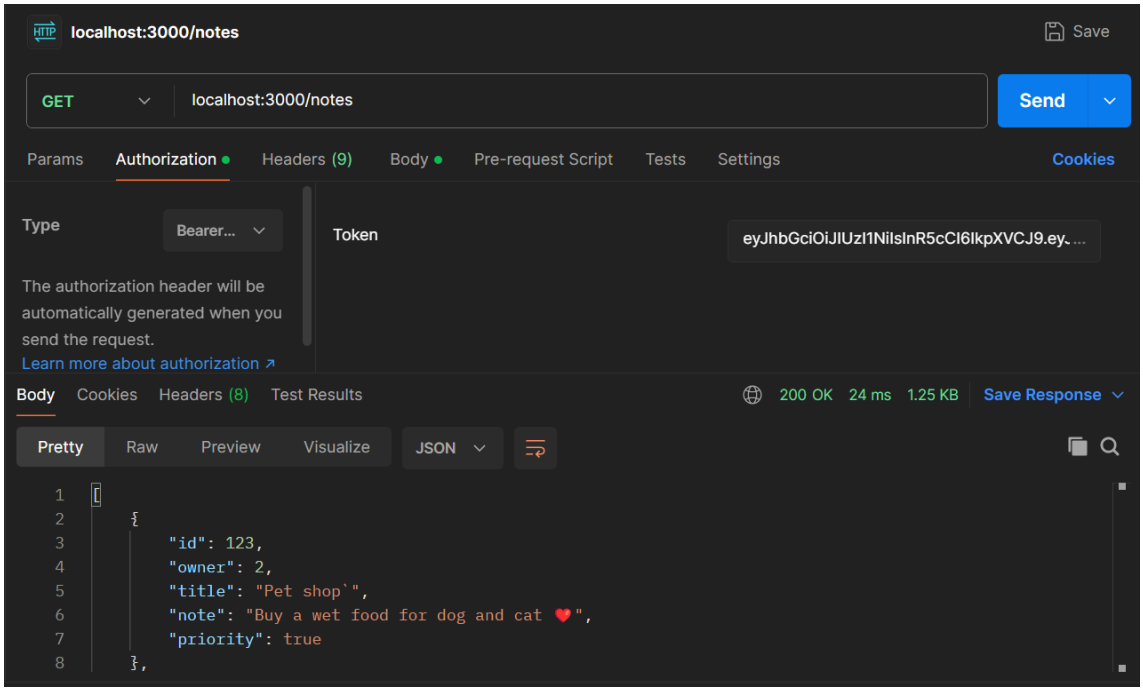
3.3.3 SSRF

W ramach dalszych testów aplikacji skupiono się na podatności SSRF, która może wystąpić w przypadku, kiedy atakujący manipuluje sposobem działania serwera i zmusza go do wykonania złośliwych żądań HTTP. Tego rodzaju ataki są niezwykle niebezpieczne, ponieważ umożliwiają atakującemu dostęp do wrażliwych danych, uzyskania kontroli nad serwerem lub wykonania innych, złośliwych działań związanych z wykorzystaniem funkcji serwera. Podstawowym rodzajem zabezpieczenia przed tego rodzaju podatnościami jest tworzenie “białej listy” zawierającej nazwy domen, z której aplikacja może pobierać zasoby. “Białe listy” mogą zawierać nie tylko nazwy dozwolonych źródeł, ale również funkcje, które mogą być wywoływane w aplikacji przez jej użytkowników, a także listę dopuszczalnych operacji. Na rysunku 3.14 została zaprezentowana implementacja funkcji odpowiadająca za weryfikację URL oraz zdefiniowana tablica adresów. Kod odpowiada za izolację aplikacji od potencjalnie złośliwych źródeł, ponieważ mechanizm białej listy ogranicza żądania HTTP serwera tylko do zaufanych domen. Po zaimplementowaniu funkcji przetestowano działanie aplikacji za pomocą narzędzia Postman. Rysunek 3.15 prezentuje odpowiedź serwera w sytuacji, gdy określona domena znajduje się na białej liście, natomiast kolejny – Rysunek 3.16 przedstawia reakcję, kiedy żądana domena nie jest w niej uwzględniona. W tym przypadku braku obecności żądanej domeny, serwer zwraca odpowiedź z kodem 403 “Forbidden”, sygnalizuje to brak zezwolenia na dostęp do danego zasobu.

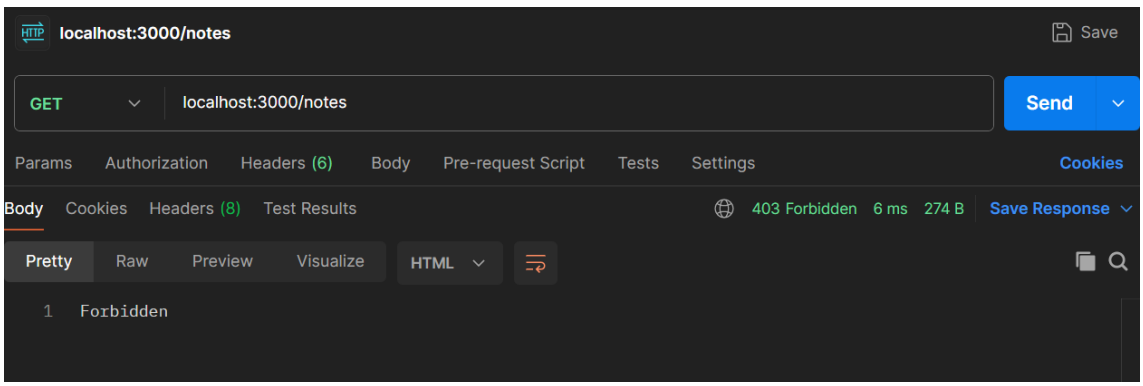
```
const whiteList = ['localhost:3000'];

const verifyURL = (req, res, next) => {
    const requestHost = req.hostname;
    if (whiteList.includes(requestHost)) {
        next();
    } else {
        res.status(403).send('Forbidden');
    }
};
```

Rys. 3.14. Funkcja weryfikująca adresy URL oraz zdefiniowana “whitelista” [opracowanie własne]

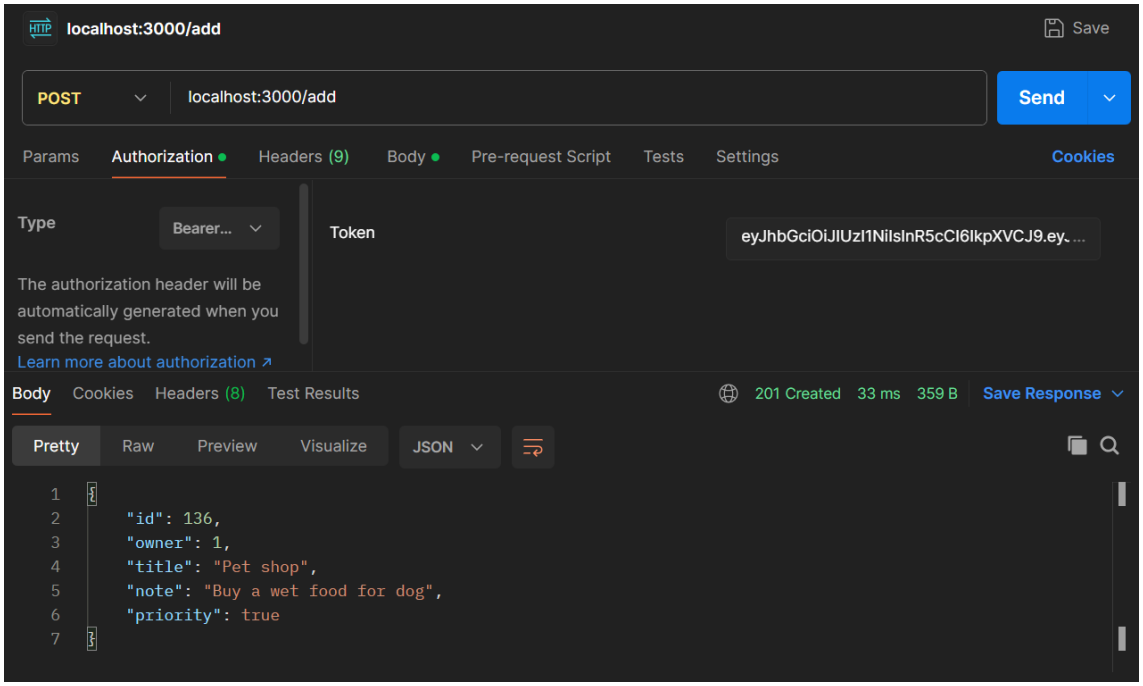


Rys. 3.15. Odpowiedź serwera - domena znajduje się na liście [opracowanie własne]

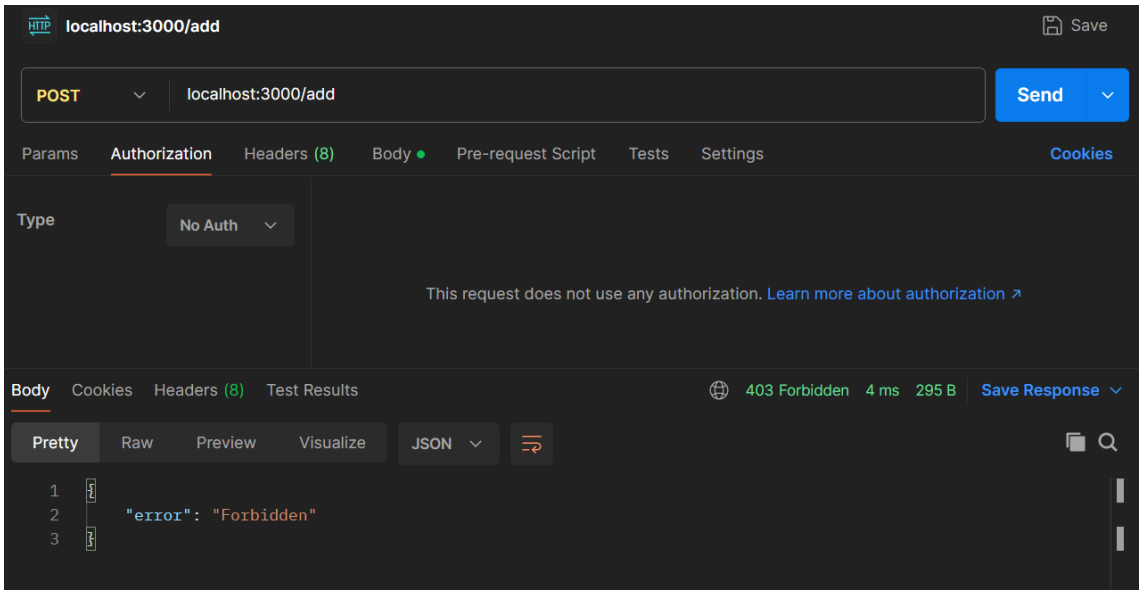


Rys. 3.16. Odpowiedź serwera - brak domeny w liście [opracowanie własne]

W ramach dalszych testów oraz implementacji odpowiednich zabezpieczeń aplikacji została rozszerzona o funkcję logowania i rejestracji użytkownika. Warunkiem dodania nowego zadania jest wcześniejsze uwierzytelnienie się użytkownika w systemie. Dostęp do funkcji wyświetlenia, dodawania oraz edytowania zadań, dostępny jest wyłącznie dla zalogowanych. Odpowiedzi serwera, kolejno dla zalogowanego i niezalogowanego użytkownika zostały zaprezentowane na rysunkach 3.17, 3.18.



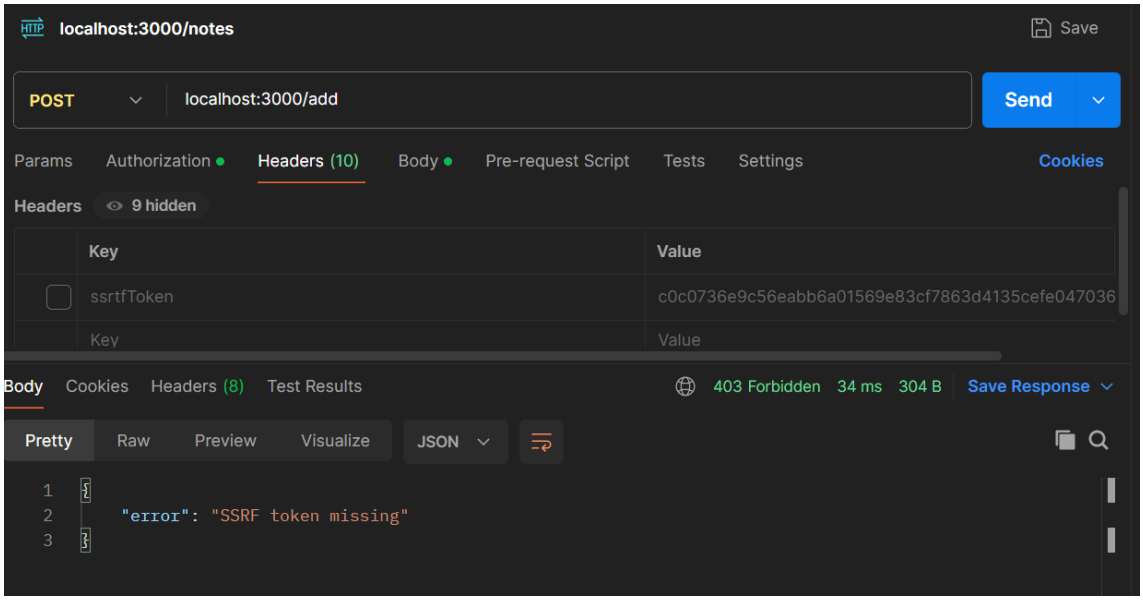
Rys. 3.17. Próba dodania nagłówka przy obecności ”Bearer token” [opracowanie własne]



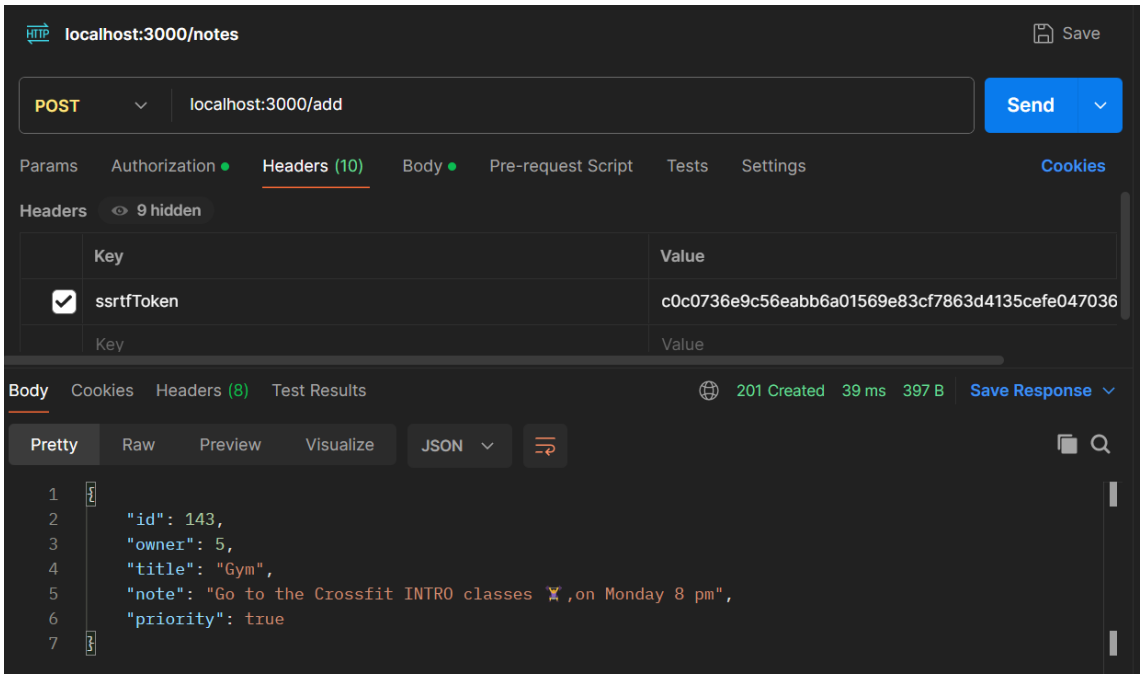
Rys. 3.18. Próba dodania nagłówka bez ”Bearer token” [opracowanie własne]

W celu dodatkowego zabezpieczenia aplikacji zaimplementowano funkcję generującą, dodatkowy token SSRF, którego celem jest weryfikowanie źródła przesyłanego żądania.

Token generowany jest przez serwer i wysyłany do użytkownika, który podczas wysyłania żądania przekazuje go z powrotem, w nagłówku. Serwer po otrzymaniu żądania od klienta weryfikuje jego zgodność i zezwala na dodanie zadania do listy. W przeciwnym wypadku, jeśli token jest nieprawidłowy, odrzuca żądanie i zwraca błąd. Rysunek 3.19 przedstawia nieudaną próbę dodania zadania, przy obecności odpowiednich nagłówków, natomiast Rysunek 3.20 prawidłowe dodanie.



Rys. 3.19. Nieudana próba dodania zadania przy braku tokenu SSRF [opracowanie własne]



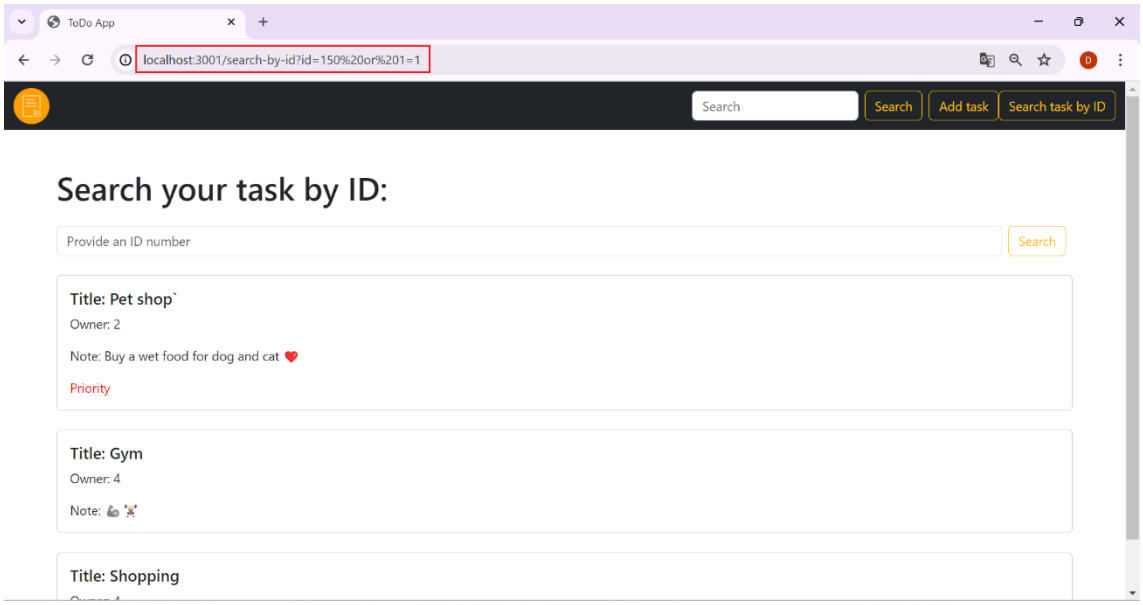
Rys. 3.20. Udana próba dodania zadania przy obecności dwóch odpowiednich tokenów [opracowanie własne]

Implementacja mechanizmu zarządzającego tokenem SSRF stanowi dodatkową warstwę zabezpieczenia aplikacji, dzięki niemu system jest bardziej odporny na ataki i wszelkie próby manipulacji żądaniami HTTP. Dodatkowo token SSRF z punktu widzenia całego systemu zapewnia elastyczność pod względem dostępu do zasobów oraz manipulacji nimi. Wprowadzenie powyższego mechanizmu w połączeniu z "białą listą" stanowi skuteczną ochronę w kontekście wzmocnienia bezpieczeństwa aplikacji oraz kontroli uprawnień użytkowników w aplikacji.

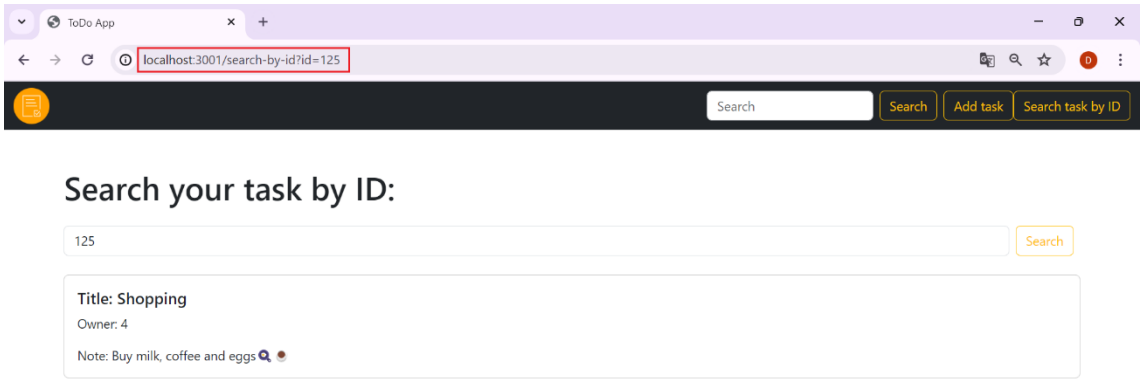
3.3.4 HPP

W wersji podstawowej aplikacja nie posiada funkcji walidacji parametrów, które przekazywane są do aplikacji z wykorzystaniem adresu URL. Brak tego rodzaju zabezpieczenia tworzy podatność umożliwiającą przeprowadzenie ataku HPP. Polega on na manipulacji parametrami w adresie URL, skutkiem tego rodzaju ataku może być uzyskanie informacji, do których nie powinniśmy mieć dostępu lub wywołanie błędu, który doprowadzi do niepoprawnego działania aplikacji. W pracy badawczej zastosowano podejście skoncentrowane manipulacji parametrami URL, która spowoduje wyświetlenie całej listy zadań, wcześniej dodanych do bazy danych. W ramach testów bezpieczeństwa przeprowadzono atak HPP, którego celem było uzyskanie dostępu do wszystkich informacji znajdujących się w konkretnej tabeli. Atak ten polegał na manipulacji parametrami linku URL, służącego do wyświetlenia strony z zadaniem, o konkretnym numerze id. W celu skutecznego przeprowadzenia ataku zmodyfikowano parametr zmieniając jego oryginalną wartość na zapytanie bazodanowe. Na rysunku 3.21 przedstawiono zmodyfikowany adres URL oraz skutek skutecznie przeprowadzonego ataku. Natomiast na kolejnym obrazku 3.22, przedstawiono widok właściwie działającej

aplikacji.



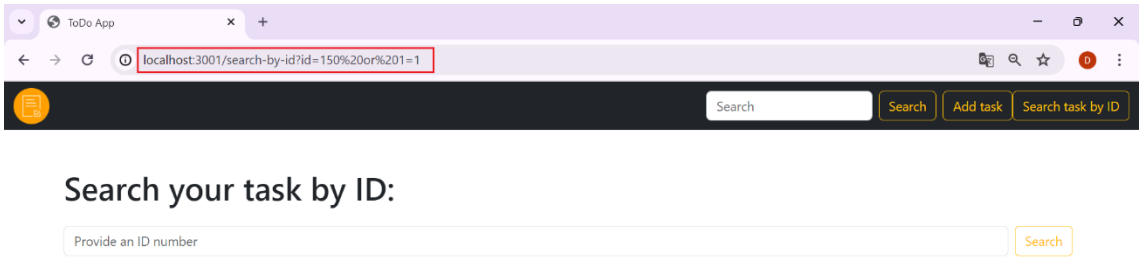
Rys. 3.21. Widok aplikacji po skutecznie przeprowadzonym ataku HPP [opracowanie własne]



Rys. 3.22. Widok poprawnie działającej aplikacji [opracowanie własne]

Aplikacja została zabezpieczona poprzez zaimplementowanie biblioteki umożliwiającej walidację parametrów przekazywanych z adresu URL. W ten sposób zapewniono, że tylko wartości numeryczne są akceptowane. Parametry zawarte w adresie URL zostają przekazane do backendu aplikacji, gdzie zostają walidowane przez program. W pierwszej kolejności sprawdzany jest ich typ, jeśli wykryta zostaje inna wartość niż liczbowa to żądanie zostaje odrzucone, gdyż nie spełnia określonego warunku. Natomiast jeśli w adresie znajdzie się dopuszczalna wartość numeryczna, następuje wysłanie zapytania do bazy danych i wyświetlenie istniejącego rekordu. Zaimplementowanie walidacji adresu

URL powoduje odrzucenie wartości innych niż numeryczne, a w konsekwencji użytkownikowi ponownie wyświetla się strona do wyszukiwania zadania, widok aplikacji został przedstawiony na rysunku 3.23. Podejście to jest wystarczającym i jednocześnie skutecznym zabezpieczeniem przed atakami typu HPP. Odpowiednio walidowane dane wejściowe mogą uchronić nie tylko przed atakami HPP, ale również zabezpieczają przed XSS, CSRF oraz wstrzyknięciem złośliwego kodu SQL. Na rysunku 3.24 został przedstawiony fragment kodu, który odpowiada za walidację parametrów URL. Dobrą praktyką jest również stosowanie walidacji danych w formularzach internetowych, po stronie użytkownika, podwójna walidacja nie tylko wpłynie na zwiększenie poziomu bezpieczeństwa, ale również odciąży prace serwera.



Search your task by ID:

Provide an ID number

Search

Rys. 3.23. Widok aplikacji po wcześniejszej implementacji zabezpieczeń i ataki HPP [opracowanie własne]

```
app.get('/notes/:id', async (req, res) => {
  const { id } = req.params;
  const { DataTypes } = require('sequelize');
  const sequelize = require('sequelize');

  const Model = sequelize.define('Model', {
    numberColumn: {
      type: DataTypes.INTEGER,
      validate: {
        isNumeric: true,
      },
    },
  })
```

Rys. 3.24. Fragment kodu odpowiadający za walidację parametrów w adresie URL [opracowanie własne]

Tabela 3.1 przedstawia wszystkie podatności aplikacji, na których skupiono się w pracy wraz ze sposobami ich zabezpieczeń. Zastosowanie ORM, "DOMPurify", walidacja danych wejściowych, podwójna weryfikacja użytkownika oraz białe listy zostały potwierdzone jako skuteczne praktyki w zwalczaniu tych zagrożeń.

Tabela 3.1. Efektywność wybranych zabezpieczeń w stosunku do danej podatności

Podatność	**Zabezpieczenie**	**Skuteczność**	**Wskazówka dotycząca dobrych praktyk**
Wstrzyknięcie SQL	ORM	Potwierdzona	Parametryzacja zapytań i stosowanie ORM
XSS	Implementacja "DOMPurify"	Potwierdzona	Użycie bibliotek do oczyszczania danych wejściowych
HPP	Walidacja danych wejściowych	Potwierdzona	Poprawna walidacja danych wejściowych
SSRF	Implementacja podwójnej weryfikacji użytkownika, białe listy	Potwierdzona	Podwójna weryfikacja użytkownika, stosowanie białych list

4. Podsumowanie

Na podstawie przeprowadzonych badań potwierdzono skuteczność zaimplementowanych zabezpieczeń oraz wyciągnięto wnioski dotyczące efektywnych praktyk zapewnienia bezpieczeństwa aplikacjom formularzowym Przedstawione podatności - wstrzyknięcie złośliwego kodu SQL, XSS, HPP oraz SSRF, stanowią poważne zagrożenie dla aplikacji formularzowych. Skuteczne przeprowadzenie ataku może doprowadzić do niepoprawnego działania aplikacji lub utraty wrażliwych danych użytkowników. Z punktu widzenia twórców oprogramowania istotną kwestią jest implementacja odpowiednich zabezpieczeń, które chronią przed zagrożeniami. Parametryzacja zapytań oraz stosowanie ORM skutecznie zabezpiecza przed atakami wstrzyknięcia złośliwego kodu SQL. Stosowanie odpowiednich bibliotek, które oczyszczają dane wejściowe z niebezpiecznych elementów, takich jak: “DOMPurify” gwarantuje odporność na ataki XSS. Dodatkowo implementowanie mechanizmu podwójnej weryfikacji użytkownika oraz stosowanie “białych list” stanowi dodatkowe zabezpieczenie w kwestii ataków typu SSRF. Odpowiednia walidacja danych wejściowych jest również ważnym elementem w kontekście ataków HPP, ponieważ stanowi to zabezpieczenie przed ewentualną próbą manipulacji parametrów w adresach URL. Wszystkie powyższe techniki zabezpieczeń, które zostały zaimplementowane w pracy stanowią ważny oraz skuteczny element w kwestii ochrony aplikacji. Oprócz wdrażania odpowiedneich zabezpieczeń, konieczne jest regularne aktualizowanie oprogramowania oraz istniejących środków ochrony w celu zapobiegania potencjalnym lukom bezpieczeństwa.

Literatura

1. OWASP, “About OWASP”: https://owasp.org/about/ (data dostępu 15.04.2024)
2. OWASP, “OWASP WebGoat”: https://owasp.org/www-project-webgoat/ (data dostępu 17.04.2024)
3. OWASP, “OWASP Top 10”: https://owasp.org/www-project-top-ten/ (data dostępu 15.04.2024)
4. Clarke-Salt Justin *SQL Injection Attacks and Defense, 2 nd edidtion,* Syngress, 2012
5. Hoffman Andrew, *Bezpieczeństwo nowoczesnych aplikacji internetowych,* Wydawnictwo: Helion, wydanie II, 2020, s. 133-144
6. Zakir Anas, *Cybersecurity & Digital Forensics,* Wydawnictwo: Clever Fox Publishing, 17.03.2022
7. Baloch Rafay, *Ethical Hacking and Penetration Testing Guide,* Wydawnictwo: CRC Press, 29.09.2017
8. Badve Sourabh, *The Hack 2.0, Wydawnictwo:* Orangebooks Publication, 2019
9. Hoffman Andrew, *Web Application Security,* Wydawnictwo: O'Reilly Media, *2024*
10. Ltd Cybellium, *Mastering OWASP*, 6.09.2023
11. Malcolm McDonald, *Web Security for Developers,* Wydawnictwo: No Starch Press, 19.06.2020
12. Maximilian Schwarzmuller, *React: kluczowe koncepcje. Przewodnik po najważniejszych mechanizmach biblioteki React, W*ydawnoctwo: Helion, 2023
13. Micah Godbolt, Frontend Architecture for Design Systems, Wydawnictwo: O'Reilly Media, 02.2016
14. Stoyan Stefanov, *React w działaniu. Tworzenie aplikacji internetowych,* Wydawnictwo: Helion*,* wydanie II, 14.02.2023, s. 97-113
15. Gantara Sagar, *React Router Quick Start Guide. Routing in React applications made easy*, Wydawnictwo: Packt Publishing, 2018
16. “DOMPurify”: https://www.npmjs.com/package/dompurify (data dostępu 23.04.2024)
17. “What is React Helmet”: https://www.ohmycrawl.com/react/helmet/#react-helmet-memory-leaks (data dostępu 23.04.2024)
18. Seqelize: https://sequelize.org/ (data dostępu 4.05.2024)
19. Oprea Ana, *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems,* Wydawnictwo: O'Reilly Media, 12.12.2022
20. Banks Alex, Porcello Eve, Learning React: Functional Web Development with React and Redux, Wydawnictwo: SHROFF, 15.06.2017

Spis rysunków

Spis Tabel