

**Politechnika Wrocławskas  
Wydział Informatyki i Telekomunikacji**

---

Kierunek: CBE

Specjalność: CBS

**PRACA DYPLOMOWA  
INŻYNIERSKA**

**Stanowisko do badania podatności systemu  
komunikatora internetowego**

Stand for testing the vulnerability of the instant messaging system

Rafał Pawłowski

Opiekun pracy  
**mgr inż. Jarosław Janukiewicz**

Słowa kluczowe: bezpieczeństwo aplikacji internetowych, programowanie, OWASP, REST

---

WROCŁAW 2022



## **Streszczenie**

Tematem pracy jest bezpieczeństwo aplikacji internetowych. Praca zostanie podzielona na dwie części – teoretyczną oraz praktyczną. W pierwszej części analizie zostaje poddana architektura nowoczesnych aplikacji webowych pod kątem ich bezpieczeństwa, przedstawiane są jedne z najbardziej szkodliwych i najpopularniejszych podatności na aplikacje internetowe oraz wybrane sposoby na zabezpieczenie przed nimi. Ataki, o których mowa oraz które zostały zawarte w pracy to kolejno *XSS*, *CSRF*, *SQL Injection*, *DoS* oraz *Third-party Dependency Injection*, a także różne warianty tych ataków. W części teoretycznej pracy zostaje także poruszony temat nowoczesnych mechanizmów uwierzytelniania oraz autoryzacji. W praktycznej części pracy natomiast została zaprogramowana aplikacja typu komunikator internetowy zbudowana według nowoczesnych oraz najlepszych praktyk dotyczących bezpieczeństwa. Aplikacja jest aplikacją internetową, kontrolowaną przez język *JavaScript* na framework'u *Vue.js*, a źródłem jej danych jest serwer napisany w technologii *Node.js* odseparowany od strony klienta (*REST API*). Aplikacja pozwala na utworzenie konta oraz zalogowanie się w celu jej użytkowania. Ponadto można dokonać autoryzacji za pośrednictwem *Google OAuth 2.0*. W aplikacji użytkownik może komunikować się z innymi osobami w wybranych publicznych pokojach lub też w prywatnej wiadomości za pomocą protokołu dwukierunkowej wymiany danych - *WebSocket*. Ponadto możliwa jest komunikacja głosowa oraz wizualna z wybranymi użytkownikami przy użyciu komunikacji w czasie rzeczywistym *WebRTC*. Następnym krokiem, po wykonaniu aplikacji, jest jej odpowiednie przetestowanie pod kątem bezpieczeństwa oraz sprawdzenie czy aby na pewno jej architektura jest na tyle dobrze zaprojektowana, że nie będzie większych problemów z jej zabezpieczeniem według głównego założenia pracy. Ataki są głównie wykonywane w przeglądarce, za pomocą wcześniej przygotowanych specjalnych skryptów oraz za pomocą maszyny wirtualnej *Kali Linux*, w celu przeprowadzenia ataku *SYN flood*. Do testów wybrano pięć wyselekcjonowanych rodzajów ataków, omawianych wcześniej w teoretycznej części pracy. Są to odpowiednio ataki *XSS*, *CSRF*, *SQL Injection*, *DoS* oraz *Third-party Dependency Injection*. Po wykryciu podatności na poszczególne ataki, jeśli w ogóle, zostały podjęte odpowiednie kroki zabezpieczające dane luki bezpieczeństwa. Sposoby obrony przed atakami były wzorowane na tych przedstawionych w teoretycznej części pracy. Po zabezpieczeniu podatności ponownie zaatakowano aplikację w ten sam sposób w celu sprawdzenia skuteczności mechanizmów obronnych. Na koniec zebrano wyniki i poddano je analizie oraz skonfrontowano z dobrymi praktykami bezpieczeństwa aplikacji webowej zawartymi w opracowaniu. Ponadto podano rekomendacje dotyczące usprawnień bezpieczeństwa aplikacji w przypadku dalszego jej rozwoju.

## **Abstract**

The subject of the thesis is the security of web applications. The study will be divided into two parts - theoretical and practical. In the first part, the architecture of modern web applications is analyzed in terms of their security, presenting some of the most harmful and popular vulnerabilities to web applications and selected methods to protect against them. The attacks in question and included in the paper are *XSS*, *CSRF*, *SQL Injection*, *DoS* and *Third-party Dependency Injection*, as well as various variants of these attacks. The theoretical part of the work also deals with the topic of modern authentication and authorization mechanisms. In the practical part of the work, an instant messaging application was programmed, built according to modern and best security practices. The application is

a web application, controlled by *JavaScript* on the *Vue.js* framework, and the source of its data is a server written in the *Node.js* technology, separated from the client's side (*REST API*). The application allows you to create an account and log in to use it. In addition, you can authorize via *Google OAuth 2.0*. In the application, the user can communicate with other people in selected public rooms or in a private message using the two-way data exchange protocol - *WebSocket*. In addition, voice and visual communication with selected users is possible using real-time *WebRTC* communication. The next step, after the application is completed, is to test it properly in terms of security and check whether its architecture is designed so well that there will be no major problems with securing it according to the main assumption of the work. Attacks are mainly performed in the browser, using pre-prepared special scripts and using the *Kali Linux* virtual machine to carry out a *SYN flood* attack. Five selected types of attacks, discussed earlier in the theoretical part of the study, were selected for the tests. These are *XSS*, *CSRF*, *SQL Injection*, *DoS* and *Third-party Dependency Injection* attacks, respectively. After detecting the vulnerability to individual attacks, if any, appropriate steps were taken to protect the data of the security vulnerability. The methods of defense against attacks were modeled on those presented in the theoretical part of the work. After the vulnerability was secured, the application was attacked again in the same way to test the effectiveness of the implemented defense mechanisms. Finally, the results were collected and analyzed and confronted with good web application security practices contained in the study. In addition, recommendations were given for improving the security of the application in the case of its further development.

## **Spis treści**

<b>1</b>	<b>Wstęp</b>	<b>6</b>
<b>2</b>	<b>Cel pracy</b>	<b>7</b>
<b>3</b>	<b>Architektura nowoczesnych aplikacji webowych</b>	<b>8</b>
3.1	Nowoczesne a tradycyjne aplikacje webowe	8
3.2	REST APIs	9
3.3	Systemy uwierzytelniania i autoryzacji	10
3.3.1	Zarządzanie autoryzacją	10
3.3.2	Niebezpieczeństwa JSON Web Token (JWT)	11
3.3.3	Zalety i wady OAuth 2.0 z perspektywy bezpieczeństwa	12
3.4	Zewnętrzne zależności aplikacji	16
3.5	Analiza architektury aplikacji	19
<b>4</b>	<b>Potencjalne zagrożenia aplikacji webowych</b>	<b>20</b>
4.1	Cross-Site Scripting (XSS)	20
4.2	Cross-Site Request Forgery (CSRF)	24
4.3	SQL Injection	28
4.4	Denial of Service (DoS)	30
4.5	Luki bezpieczeństwa zależności zewnętrznych	34
<b>5</b>	<b>Obrona przed atakami w sieci</b>	<b>36</b>
5.1	Bezpieczeństwo współczesnych aplikacji	36
5.2	Ochrona przed atakami XSS	37
5.3	Ochrona przed atakami CSRF	41
5.4	Ochrona przed SQL Injection	43
5.5	Przeciwdziałanie DoS	44
5.6	Zabezpieczanie zewnętrznych zależności	46
<b>6</b>	<b>Testy zastosowanych zabezpieczeń</b>	<b>48</b>
6.1	Wystawienie serwisów aplikacji do sieci lokalnej	48
6.2	Testowanie aplikacji	48
6.2.1	Przeprowadzenie ataków	48
6.3	Analiza wyników oraz dalsze działania	68
<b>7</b>	<b>Podsumowanie</b>	<b>70</b>
	<b>Literatura</b>	<b>72</b>

## 1. Wstęp

Bezpieczeństwo w sieci od dłuższego czasu jest powszechnym tematem. W dzisiejszych czasach dostęp do Internetu oraz aplikacji internetowych ma niemalże 5 mld osób na całym świecie. Stanowi to wręcz idealną płaszczyznę dla hakerów do przeprowadzania ataków na ogromne skale. Już nie raz w historii, przez nieodpowiednie zabezpieczenia, wiele aplikacji internetowych zostało ofiarami ataków, wskutek czego doszło do wycieków danych oraz utraty użytkowników. W związku z tymi wydarzeniami, bezpieczeństwo w sieci zaczęło stawać się coraz poważniejszym tematem, gdyż koszty zapobiegnięcia problemom są znacznie niższe niż koszty odbudowy po dokonanym ataku. Poruszana w temacie pracy aplikacja internetowa typu komunikator jest świetnym miejscem do testowania bezpieczeństwa w sieci, ponieważ w tego typu aplikacjach należy zwrócić uwagę na wiele płaszczyzn ataków, w tym: uwierzytelnienie, autoryzacja, szyfrowanie danych itd. Dodatkowo, w dobie pandemii, popyt na tego typu aplikacje znacznie wzrósł w stosunku do ubiegłych lat, dlatego też, ze względu na większy ruch użytkowników w komunikatorach internetowych, jest to świetne miejsce do przeprowadzania różnych rodzajów ataków przez hakerów.

W dzisiejszych czasach aplikacje pisane są w sposób znacznie bezpieczniejszy niż to odbywało się dekadę bądź dwie temu, gdyż nacisk na bezpieczeństwo jest bardzo wysoki. Ponadto wiele nowoczesnych bibliotek i frameworków wymusza także na programistach pisanie bezpiecznego kodu. Dlatego też niniejsza praca będzie opierała się na zbadaniu architektury nowoczesnych aplikacji internetowych i sprawdzeniu czy aby na pewno jej założenia prowadzą do bezpiecznego produktu końcowego.

Współczesne metody uwierzytelniania i autoryzacji są również istotnym tematem, na który na pewno należy zwrócić uwagę, ponieważ każdy użytkownik Internetu ma z nimi (uwierzytelnieniem i autoryzacją) do czynienia na co dzień. Dlatego, przy budowie aplikacji, ważne jest, aby dobrać odpowiednie metody uwierzytelnienia oraz autoryzacji – najlepiej te nowoczesne oraz sprawdzone – oraz przetestować ich działanie pod kątem bezpieczeństwa.

W pracy zostaną również przedstawione, staranie wyselekcjonowane przez autora, jedne z najpopularniejszych oraz najniebezpieczniejszych ataków na aplikacje internetowe. Następnie przedstawione zostaną odpowiednio sposoby na zabezpieczanie się przed wcześniej wspomnianymi atakami. Bez wiedzy o niebezpieczeństwach w sieci oraz sposobów obrony przed nimi, trudnym jest poruszać się po Internecie oraz projektować aplikacje internetowe bez natknięcia się na niebezpieczeństwa.

Po przedstawieniu odpowiednich założeń teoretycznie bezpiecznej aplikacji internetowej, koniecznym będzie zaprojektowanie realnej aplikacji typu komunikator internetowy w sposób nowoczesny oraz sprawdzenie jej pod kątem bezpieczeństwa. Przeprowadzony zostanie szereg testów aplikacji, a następnie zebrane wyniki zostaną skonfrontowane z wcześniej przedstawionymi w pracy założeniami. Wyniki te dowiodą bądź nie, czy aby na pewno architektura nowoczesnych aplikacji pozwala na budowanie ich w sposób bezpieczny.

Praca będzie bazować głównie na źródłach [1], [2], [3] oraz na wiedzy autora, gdyż ten posiada już podstawową wiedzę na temat zabezpieczania aplikacji internetowych.

## **2. Cel pracy**

Celem pracy jest wdrożenie stanowiska do badań podatności systemu komunikatora internetowego na potrzeby zajęć w laboratorium cyberbezpieczeństwa. Przeprowadzanie badań na takim stanowisku pozwoli na zwiększenie świadomości odbiorcy na jakie niebezpieczeństwa wystawiany jest w sieci. Po zapoznaniu się z treścią pracy, słuchacz powinien znać najpopularniejsze a zarazem jedne z najniebezpieczniejszych ataków webowych oraz wiedzieć jak przed nimi skutecznie zabezpieczać swoją aplikację. Praca zostanie podzielona na dwie części – teoretyczną oraz praktyczną. W pierwszej części zostaną podjęte analizy architektur nowoczesnych aplikacji webowych pod kątem ich bezpieczeństwa, przedstawione zostaną jedne z najbardziej szkodliwych podatności oraz wybrane sposoby na ochronę przed nimi. Bardziej wyrafinowane szczegóły oraz listy popularnych ataków wraz z przeróżnymi metodami obrony przed nimi można znaleźć tutaj [4]. Na drugą część pracy – praktyczną – będzie składać się samodzielne napisanie aplikacji internetowej typu komunikator oraz przetestowanie jej pod kątem bezpieczeństwa. Aplikacja będzie działać w przeglądarce, a źródłem danych będzie serwer napisany w technologii *Node.js* odseparowany od strony klienta (*REST API*). Aplikacja będzie pozwalała na utworzenie konta oraz zalogowanie się w celu jej użytkowania. Ponadto będzie można dokonać autoryzacji za pośrednictwem *Google*. W aplikacji użytkownik będzie mógł komunikować się z innymi osobami w wybranych publicznych pokojach lub też w prywatnej wiadomości za pomocą protokołu dwukierunkowej wymiany danych - *WebSocket*. Ponadto możliwa będzie komunikacja głosowa oraz wizualna z wybranymi użytkownikami przy użyciu komunikacji w czasie rzeczywistym *WebRTC*. Po wykonaniu aplikacji zostaną przeprowadzone testy pod kątem bezpieczeństwa. Zebrane wyniki zostaną poddane analizie oraz skonfrontowane z dobrymi praktykami bezpieczeństwa aplikacji webowej zawartymi w opracowaniu.

### **3. Architektura nowoczesnych aplikacji webowych**

W niniejszym rozdziale zostanie omówiona struktura nowoczesnych aplikacji webowych. Rozdział ten składa się z pięciu podrozdziałów, w których zostaną porównane budowy starszych oraz nowoczesnych aplikacji, sprawdzona zostanie współczesna, jedna z najpopularniejszych architektur serwera – *REST API*, oraz dokona się przeglądu mechanizmów uwierzytelnienia oraz autoryzacji sprostującym dzisiejszym wymaganiom. Ponadto zostanie sprawdzony wpływ zewnętrznych zależności aplikacji, takich jak biblioteki czy frameworki, na jej bezpieczeństwo oraz zostaną poddane analizie jej słabe punkty.

#### **3.1. Nowoczesne a tradycyjne aplikacje webowe**

Szybki postęp w dziedzinie budowania oraz projektowania aplikacji webowych sprawił, że przepaść technologiczna między nowoczesnymi aplikacjami, a tymi projektowanymi dekadę temu jest ogromna. Kiedyś aplikacje osadzane były głównie na serwerze który renderował pliki *HTML/Javascript/CSS* i wysyłał je do klienta za każdym jego żądaniem. Miało to jedną poważną wadę – strona musiała przeładowywać się za każdym razem, gdy dynamicznie chcieliśmy zmienić czy pobrać jej zawartość. Niedługo po tym zaczęto coraz częściej wykorzystywać protokół *HTTP* do wymiany danych pomiędzy serwerem a klientem co znacznie zmniejszyło liczbę przeładowań aplikacji.

Dzisiaj aplikacje zazwyczaj są reprezentowane przez dwie lub więcej aplikacji komunikujących się ze sobą przez protokół sieciowy, wspólnie połączonych poprzez *REST API*. Aplikacje te nie utrzymują stałego połączenia z klientem (nie przechowują żadnych informacji o kliencie), a jedynie wysyłają pakiety danych na jego żądanie. Działanie nowoczesnych aplikacji webowych można z czystym sumieniem porównać do działania programów natywnych pod względem tego, jak samodzielnie potrafią zarządzać własnymi pętlami cyklu życia, żądać potrzebnych im danych oraz nie wymagają odświeżenia strony po uprzednim, wstępny załadunku. Ponadto współczesna aplikacja internetowa charakteryzuje się separacją strony klienta oraz serwera (bądź też serwerów, ponieważ możemy mieć wiele niezależnych od siebie serwisów zespolonych ze sobą za pomocą architektury *REST API*), co znaczaco wpływa na bezpieczeństwo tejże aplikacji. [1]

Przeciętna, współczesna aplikacja internetowa prawdopodobnie wykorzystuje kilka z podanych poniżej technologii:

- *REST API*
- *JSON*
- *JavaScript*, opcjonalnie na frameworki (*React.js*, *Vue.js*) [5]
- System uwierzytelniania i autoryzacji
- Jeden lub kilka serwerów (najczęściej na serwerze *Linux*)
- Jedna lub kilka baz danych (*MySQL*, *Redis*, *Firebase*)
- Miejsce do przechowywania danych po stronie klienta (*cookies*, *local storage*)

Oczywiście technologie, które są na co dzień używane, jest znacznie więcej, ponieważ przeglądarki zdążyły zaadaptować się do dzisiejszych wymagań i oferują znacznie więcej możliwości niż przykładowo dekadę temu, jednakże wymienione zostały te najbardziej podstawowe oraz niemalże każdorazowo wykorzystywane. Wzrostające możliwości oraz coraz to nowsze technologie niosą ze sobą nowe luki w zabezpieczeniach, które można w różny (niestety niepożądany też) sposób wykorzystać. Dlatego w dzisiejszych czasach tak bardzo należy uważać na podatności czyniące w Internecie.

### 3.2. REST APIs

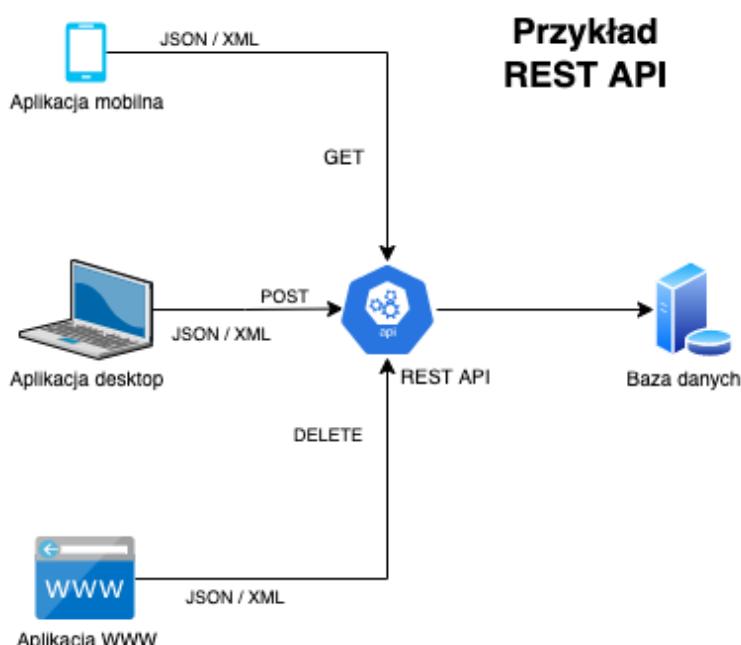
Przed omówieniem czym dokładnie *REST (Representational State Transfer)* jest, warto na początku podać kilka jego unikalnych cech.

Pierwszą z nich jest bez wątpienia oddzielność od strony klienta. *REST APIs* zostały zaprojektowane tak, aby tworzyć wysoce skalowalne a jednocześnie nieskomplikowane aplikacje. To wszystko sprawia, że klient nie musi się martwić o wykonywanie zapytań do bazy czy wykonywanie logiki serwerowej, gdyż to wszystko za niego robi oddzielnie serwer (bądź też serwery).

Kolejną cechą tej architektury jest bezstanowość. Serwer nie może przetrzymywać jakichkolwiek danych o stronie klienta. Jego zadaniem jest jedynie przetwarzanie żądań i odsyłanie odpowiedniej odpowiedzi (danych). Nie oznacza to jednak niemożności dokonywania uwierzytelnienia bądź autoryzacji przez serwer – takie informacje powinny być wysyłane w postaci tokenów na serwer.

Następną cechą *REST APIs* jest buforowalność – podstawa skalowalności aplikacji internetowej. Każdy punkt końcowy w API dokładnie określa jakie dane ma zwrócić, dlatego wprowadzenie buforowalności w ww. architekturze jest stosunkowo łatwym zadaniem. Należy jednak uważać, aby „zcacheowane” dane nie zawierały wrażliwych danych takich jak hasła czy numery kard kredytowych, by nie doprowadzić do niechcianego wycieku danych.

Na koniec, każdy punkt końcowy w architekturze *REST* powinien dokładnie definiować obiekt, do którego się odnosi, np. „/users/jan”. Wprowadzenie werbalności w nazewnictwie punktów końcowych spowoduje, że staną się one samodokumentowane, tzn. ich nazwy będą na tyle same w sobie opisowe, że nie trzeba będzie ich dodatkowo opisywać. Ponadto ww. struktura pozwoli na swobodne wykorzystywanie metod protokołu *HTTP* tj. *GET*, *POST*, *PUT*, *PATCH* czy *DELETE*, co też jest fundamentalną składową architektury *REST API*. [6] Koncepcję działania *REST API* przedstawia rysunek 3.1.



Rysunek 3.1 Przykład komunikacji urządzeń z REST API

Najczęściej używanym formatem danych, jakie są przesyłane w *REST API*, jest format typu *JSON (JavaScript Object Notation)*. Format ten wyparł niegdyś najpopularniejszy format *XML* z tego względu, że jest mniej werbalny oraz znacznie czytelniejszy od tego drugiego. Dla porównania formy obydwu formatów, zestawiono je ze sobą poniżej. [1]

```
// XML
<user>
  <name>Jan</name>
  <surname>Kowalski</surname>
  <birth>16.01.1980</birth>
  <contact-details>
    <phone>123456789</phone>
    <email>kowalski@gmail.com</email>
  <contact-details>
</user>

// JSON
{
  "name": "Jan",
  "surname": "Kowalski",
  "birth": "16.01.1980",
  "contact-details": {
    "phone": "123456789",
    "email": "kowalski@gmail.com"
  }
}
```

W przypadku potrzeby odtworzenia warstwy *API* aplikacji internetowej przy pomocy techniki inżynierii wstępnej, bardzo ważnym jest zrozumienie jak *REST-owa* architektura jest zbudowana, ponieważ znaczną część dzisiejszych aplikacji bazuje na tej właśnie strukturze. Ponadto wiele wykorzystywanych obecnie narzędzi jest udostępnianych za pomocą *REST APIs*.

### 3.3. Systemy uwierzytelniania i autoryzacji

Tak jak wcześniej zostało już wspomniane, w architekturze nowoczesnych aplikacji internetowych najczęściej mamy do czynienia z separacją strony klienta od strony serwera. W związku z tym serwer w pewien sposób musi wiedzieć czy może pozwolić na dostęp do pewnych wrażliwych, trwale zapisanych danych dla żądającego ich użytkownika.

Wobec wyżej przedstawionego problemu, pomocne będzie rozróżnienie dwóch terminów: uwierzytelnianie i autoryzacja. Pierwsza z nich odpowiada za zweryfikowanie czy klient podający się za pewną osobę – niech będzie to Jan, jest faktycznie Janem.

Autoryzacja natomiast definiuje nam to, czy wcześniej uwierzytelniony użytkownik ma dostęp do pewnej porcji danych w porównaniu do innych użytkowników. Innymi słowy określa, przykładowo, czy przysłowiowy Jan może sięgnąć po rekordy Tomasza i vice versa.

Oba wyżej wymienione systemy są bardzo ważnym elementem prawidłowego i przede wszystkim bezpiecznego działania dzisiejszych aplikacji internetowych, a jeszcze ważniejsza jest ich prawidłowa implementacja.

#### 3.3.1 Zarządzanie autoryzacją

Kolejnym krokiem, zaraz po uwierzytelnieniu, jest autoryzacja. Systemy autoryzacji są znacznie trudniejsze do skategoryzowania, ponieważ w dużej mierze zależą one od logiki biznesowej wewnętrz aplikacji internetowej.

Aplikacje napisane zgodnie z dobrymi praktykami posiadają zcentralizowany mechanizm złożony z bram oraz polityk dostępu odpowiadający za całą autoryzację w aplikacji oraz określający prawa dostępu do zasobów czy funkcjonalności przez danego użytkownika. [1]

Przykładem złego zaprojektowania oraz wdrożenia autoryzacji jest implementacja jej na każdym *API* z osobna. Takie podejście wiedzie do błędów oraz luk bezpieczeństwa w aplikacji, ponieważ, manualnie zabezpieczając każde *API* z osobna, możemy zwykle się pomylić i dać uprawnienia niechcianym użytkownikom czy też zapomnieć zaimplementować pewnych bram dostępu do danych.

Najważniejszymi zasobami, które zawsze powinny mieć zaimplementowane mechanizmy autoryzujące, są między innymi:

- edycja profilu bądź ustawień
  - reset hasła
  - odczyt oraz zapis prywatnych wiadomości
  - funkcje związane z płatnościami
  - zwiększone uprawnienia użytkownika (administrator)

### 3.3.2 Niebezpieczeñta JSON Web Token (JWT)

*JSON Web Token* to mechanizm uwierzytelnienia pozwalający nam zweryfikować „właściciela” pewnych danych w formacie *JSON*. *JWT* jest kryptograficznie podpisany, zakodowanym w sposób procentowy ciągiem znaków, który może przechowywać nieskończoną ilość danych. Ze względów bezpieczeństwa, token powinien być generowany oraz podpisywany po stronie serwera, ponieważ w takich okolicznościach serwer, w momencie otrzymania *JWT*, będzie mógł zagwarantować integralność danych, na których operuje (nieautoryzowana osoba nie będzie mogła modyfikować danych, do których ma wzbroniony dostęp). Dodatkowo *JSON Web Token* charakteryzuje się odpornością na ataki typu „*Man in the Middle*”, gdyż raz wysłany token nie może być w trakcie przetwarzania zmodyfikowany. *JWT* przedstawia się w postaci: „klucz”: „wartość”, a jego przykład, wzięty z własnej aplikacji, możemy znaleźć na Rysunku 3.2. [7]

Rysunek 3.2 Przykład JSON Web Token

Jedną z wad bezpieczeństwa JWT jest brak gwarancji poufności danych. Dane schowane w tokenie są jedynie serializowane, natomiast same z siebie nie podlegają szyfrowaniu. Aby zapewnić poufność w *JSON Web Token*, rekommendowane jest użycie protokołu *HTTPS* wraz z implementacją tego mechanizmu uwierzytelnienia oraz autoryzacji.

Dodatkową rekomendacją przy stosowaniu *JWT* jest powstrzymanie się od zapisu tych tokenów w sesji użytkownika z dwóch powodów. Pierwszym z nich jest brak możliwości usunięcia tokenu po zakończeniu sesji (wyłączeniu przeglądarki bądź wylogowaniu), gdyż *JWT* działa niezależnie, a co za tym idzie – przy opisanym problemie nie będzie możliwości, aby ten token w porę unieważnić. Takie zachowanie jest niepożądane oraz ingerujące w bezpieczeństwo aplikacji, gdyż wystawiamy nieużywany (ale ciągle ważny) token na niebezpieczeństwo przejęcia. Drugim powodem jest szeroki zakres funkcji oraz duży zasięg

JWT, co zwiększa prawdopodobieństwo pomyłek zarówno ze strony użytkowników jak i programistów czy też autorów bibliotek.

Dobrą praktyką w zwiększeniu bezpieczeństwa *JSON Web Token* jest ustawianie jego „terminu ważności” (ekspracji). Minimalizuje to ryzyko „wykradnięcia” tokenu i zwykle nie powinno być dłuższe niż 15-30 minut od momentu wygenerowania JWT. Po upływie tego czasu token dostępu powinien być odświeżony. Dzieje się to za pomocą drugiego tokenu – odświeżania (*refresh token*), którego logikę należy oddziennie zaimplementować. Jak widzimy na rysunku 3.3.2-1, przedstawione są tam dwa tokeny, odpowiednio: token odświeżania oraz token dostępu. Pierwszy z nich odpowiada za odświeżanie tego drugiego, gdy temu skończy się termin ważności. Drugi natomiast jest używany do uwierzytelnienia oraz autoryzacji.

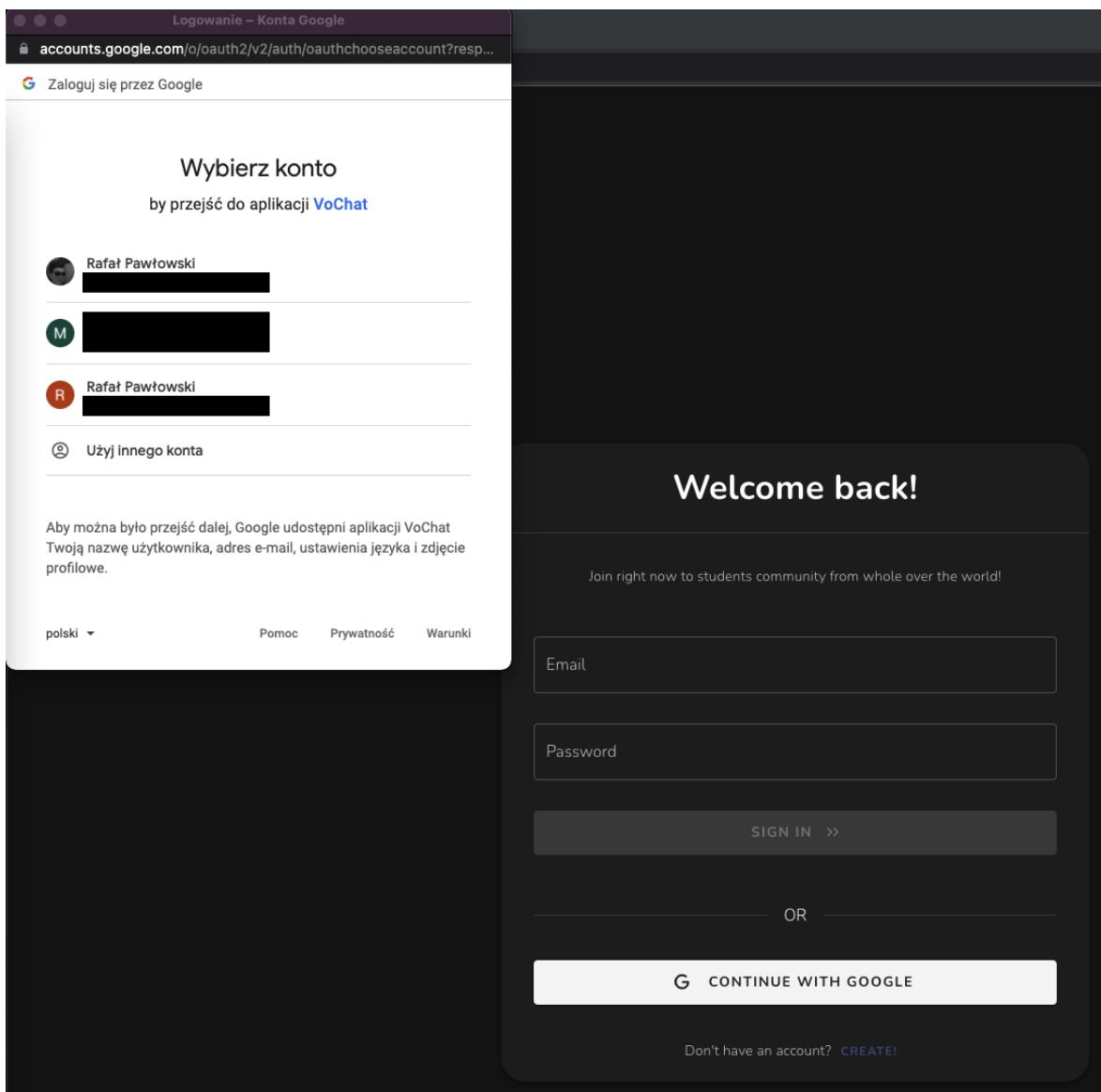
Podczas wymiany tokenu między serwerem a klientem, ważne jest także jego prawidłowe i bezpieczne przechowywanie po stronie front-endu. Jednym z bezpieczniejszych rozwiązań jest przechowywanie JWT w cookie w trybie *HTTP-only*. Jest to specjalny rodzaj „ciasteczka”, które jest wysyłane jedynie w żądaniach *HTTP* do serwera co zapobiega atakom *Cross-Site Scripting (XSS)* próbującym wykraść zawartość tokenu za pomocą skryptów JavaScript wykonywanych w przeglądarce, np. przy pomocy *document.cookie*. [8]

*JSON Web Token* jest bardzo popularnym wyborem mechanizmu uwierzytelnienia oraz autoryzacji. Należy jednak pamiętać, że nie jest on wyborem idealnym ze względów bezpieczeństwa, dlatego należy wiedzieć jak prawidłowo go zabezpieczyć przed potencjalnymi atakami.

### 3.3.3 Zalety i wady OAuth 2.0 z perspektywy bezpieczeństwa

*OAuth 2.0* to otwarty protokół, który pozwala nam na implementację bezpiecznych mechanizmów autoryzacji za pomocą różnych platform tj. aplikacji internetowych czy mobilnych. Docelowym celem tego frameworki jest oddelegowanie autoryzacji do zasobów, tzn., że właściciel określonego zasobu może pewnemu podmiotowi na określony czas przypisać dostęp. Praktycznym przykładem, jakim udało mi się zaimplementować, jest udzielenie aplikacji zewnętrznej dostępu do danych z profilu platformy *Google* (można tu użyć wielu różnych platform implementujących ten mechanizm tj. *Facebook* czy *GitHub*). Poniżej, w kilku krokach, przedstawiono scenariusz działania *OAuth 2.0*: [9]

1. Aplikacja w celu pobrania danych użytkownika z platformy, w tym wypadku *Google*, musi przekierować nas do serwera autoryzującego, na którym znajdują się informacje o naszym profilu tak jak widać to na Rysunku 3.3.



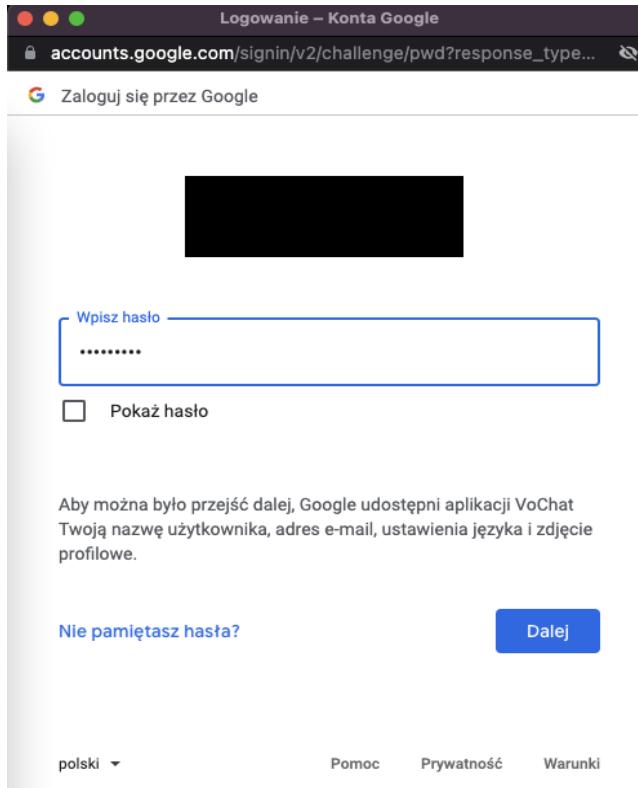
Rysunek 3.3 Okno serwera autoryzującego Google

2. Platforma, na której dokonuje się autoryzacji, informuje nas o tym, że zewnętrzna aplikacja chce uzyskać dostęp do pewnych danych naszego profilu np. email, imię czy preferowany język, jak widać na Rysunku 3.4:

**Aby można było przejść dalej, Google udostępnii aplikacji VoChat Twoją nazwę użytkownika, adres e-mail, ustawienia języka i zdjęcie profilowe.**

Rysunek 3.4 Informacja o dostępie do danych

3. Następnie, na Rysunku 3.5, przedstawiona jest próba otrzymania autoryzacji, tj. wprowadzenia poprawnych danych logowania do serwera autoryzującego oraz zgody na narzucone przez niego warunki.



Rysunek 3.5 Uwierzytelnienie konta Google

4. Gdy autoryzacja przebiegnie domyślnie, serwer autoryzujący dokona powrotnego przekierowania do naszej aplikacji przesyłając przy tym token za pomocą którego będzie mógł pobrać dane z profilu autoryzowanego użytkownika.

A screenshot of a browser window titled "app" showing a list of active users with "Marek" listed. Below the window is a screenshot of the Chrome DevTools Network tab. The tab is set to the "Application" section. A table shows two entries: "express:sess.sig" with value "p7tV5lmmerrCSA7DsA\_t6I4QW9fE" and "Domain" "localhost" and "Path" "/" with expiration "2021-10-29T21:29:..." and "HttpOnly" checked; and "express:sess" with value "eyJwYXNzcG9ydCI6eyJ1c2Vyljofz..." and "Domain" "localhost" and "Path" "/" with expiration "2021-10-29T21:29:..." and "HttpOnly" checked.

Rysunek 3.6 Pomyślna autoryzacja za pośrednictwem OAuth 2.0 od Google

Na Rysunku 3.6 przedstawione jest uzyskanie tokenu *OAuth 2.0* niezbędnego do przeprowadzenia autoryzacji. Token ten najczęściej jest wygenerowanym losowo ciągiem znaków o zadanej długości, który następnie jest sprawdzany na serwerze, za każdym żądaniem klienta do pewnych zasobów, w celu zweryfikowania do nich autoryzacji danego użytkownika.

Z punktu widzenia bezpieczeństwa aplikacji, autoryzacja z wykorzystaniem *OAuth 2.0* posiada wiele zalet. Wśród nich na pewno możemy wyodrębnić rozdzielnosć zewnętrznej aplikacji od danych uwierzytelniających użytkowników. Dzięki temu atakujący próbujący „zhackować” aplikację zewnętrzną, mają bardziej ograniczony dostęp do danych użytkowników, gdyż te znajdują się u dostawcy serwera autoryzacyjnego. Ponadto można wykorzystywać ten jeden serwer autoryzacyjny do chronienia wielu zasobów w aplikacji zewnętrznej a dzięki wykorzystaniu jednego takiego serwera w różnych aplikacjach, ogranicza się zakładanie różnych kont o podobnych albo i nawet takich samych danych uwierzytelniających.

Jednakże i to rozwiązanie nie obyłoby się bez wad. W implementacji *OAuth 2.0* całkowicie zrezygnowano z szyfrowania komunikacji, dlatego implementując ten mechanizm należy zadbać o szyfrowanie go przy pomocy protokołu *TLS*. Brak szyfrowania oznacza brak poufności danych, a co za tym idzie narażamy nasz token na bycie wykradniętym przez osoby trzecie.

Ważne jest także odpowiednie zabezpieczenie serwera autoryzującego. Przede wszystkim należy zadbać o jego odpowiednią redundancję oraz odporność na ataki typu *DoS*. Serwer powinien wiedzieć, kiedy i czy może nadać uprawnienia dla danego użytkownika. Powinien być też odporny na ataki typu brute-force, ponieważ istnieje taka techniczna możliwość, aby odgadnąć token czy dane uwierzytelniające klienta, dlatego serwer powinien być przygotowany na takie przypadki. Kolejnym błędem implementacyjnym mechanizmu *OAuth 2.0* może być niepowiązanie tokenu z danym użytkownikiem. Może to doprowadzić do tego, że teoretycznie autoryzowany użytkownik nigdy nie otrzyma praw dostępu do danych albo losowy użytkownik otrzyma je przypadkowo. Ponadto serwer autoryzacyjny powinien mieć możliwość unieważnienia tokenu (najczęściej dzieje się to po wylogowaniu użytkownika) oraz jego odświeżania, przy czym należy pamiętać, aby odświeżanie nie polegało na wydłużeniu czasu żywotności, a na unieważnieniu starego i wygenerowaniu nowego tokenu. Ponadto token powinien być przechowywany w postaci funkcji skrótu zamiast w niebezpiecznej jawnej formie. Zaimplementowany przeze mnie system do generowania, odświeżania oraz walidacji tokenów został pokazany na Rysunkach 3.7 oraz 3.8.

```
exports.generateAccessToken = user => {
  return jwt.sign(user, process.env.ACCESS_TOKEN_SECRET, {
    expiresIn: "1800s",
  });
};

exports.refreshAccessToken = async refreshToken => {
  const refreshTokens = await redis.getRefreshTokens();

  if (!Object.values(refreshTokens).includes(refreshToken)) {
    return null;
  }

  return jwt.verify(
    refreshToken,
    process.env.REFRESH_TOKEN_SECRET,
    (err, user) =>
      err ? null : exports.generateAccessToken({ name: user.name })
  );
};
```

Rysunek 3.7 Generowanie i odświeżanie JWT

```

exports.authenticateToken = (req, res, next) => {
  if (req.user) {
    return next();
  }

  const { accessToken, refreshToken } = req.cookies;

  if (!accessToken) {
    return res.sendStatus(401);
  }

  jwt.verify(
    accessToken,
    process.env.ACCESS_TOKEN_SECRET,
    async (err, user) => {
      if (err) {
        const newToken = refreshToken
          ? await utils.refreshAccessToken(refreshToken)
          : null;

        if (!newToken) {
          return res.sendStatus(403);
        }

        res.cookie(EnumTokens.ACCESS_TOKEN, accessToken, {
          maxAge: 30 * 60 * 1000,
          httpOnly: true,
        });
      }

      req.user = user;
      next();
    }
  );
}

```

Rysunek 3.8 Weryfikacja JWT po stronie serwera

W przypadku zabezpieczeń tokenu *OAuth 2.0* po stronie klienta, nie różnią się one zbytnio od tych omawianych w przypadku *JWT*. Dodatkową podatnością jaką uchyla przed nami użycie mechanizmu *OAuth 2.0* (w przypadku front-endu) jest wystawienie na atak typu *Cross-Site Request Forgery (CSRF)*. Warto dlatego upewnić się czy przekierowanie ze strony klienta do serwera autoryzującego posiada parametr typu *state* chroniący przed omawianym atakiem. Użycie tego parametru jest ukazane na Rysunku 3.9.

tenant.auth0.com/authorize?...&state=xyzABC123

Rysunek 3.9 Link zabezpieczony przed atakiem typu *CSRF*

### 3.4. Zewnętrzne zależności aplikacji

Dzisiejsze aplikacje z reguły posiadają wiele zewnętrznych zależności takich jak biblioteki, frameworki czy zintegrowane kody zewnętrzne. Pozwala to na szybsze rozwijanie aplikacji poprzez skupienie się na logice biznesowej zamiast „wynajdywaniu koła na nowo” przepisując tą samą, na co dzień używaną logikę. Ponadto, dzięki temu, istnieje możliwość integracji danej aplikacji z *API* innych firm, co również oszczędza

nadmierną pracę oraz poprawia skalowanie produktu. Jednakże użycie zewnętrznych zależności w aplikacji nie jest bez wad, ponieważ osoba trzecia, budując daną bibliotekę czy framework, mogła popełnić błąd i otworzyć lukę bezpieczeństwa tym samym narażając naszą aplikację na dokładnie taką samą podatność.

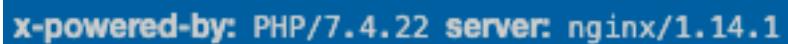
Relacje zewnętrzne po stronie klienta aplikacji są stosunkowo łatwe do wykrycia. W zależności od tego jakiej biblioteki czy jakiego framework'u używamy, istnieją przeróżne sposoby (najczęściej znajdziemy je w dokumentacji) na wystawienie wersji danej zależności w narzędziach developerskich przeglądarki. W aplikacji stworzonej na potrzeby tej pracy używany jest framework *Vue.js*, którego wersję możemy wystawić w podany niżej sposób:

```
const version = Vue.version;  
console.log(version);
```

Znając wersję zależności zewnętrznej, w łatwy sposób można określić na jakie podatności w zabezpieczeniach *ReDoS*, *Prototype Pollution* czy *XSS (Cross-Site Scripting)* jest ona wystawiona.

Wykrywanie wersji oprogramowania po stronie serwera jest znacznie bardziej skomplikowane aniżeli po stronie klienta, ponieważ, w tym drugim, kod potrzebny do funkcjonowania aplikacji jest często pobierany do pamięci przeglądarki, gdzie istnieje możliwość interakcji z nim poprzez *DOM (Document Object Model)* [10]. Strona serwera natomiast wymaga większej wiedzy o frameworku, który wymaga wykrycia. Jest to znacznie bardziej zadanie, aczkolwiek nie jest ono niemożliwe, gdyż czasami zależności serwerowe zostawiają ślady w ruchu *HTTP* (np. w postaci nagłówków) czy też ujawniają własne punkty końcowe.

Źle skonfigurowane serwery często doprowadzają do takiej sytuacji, że w wysyłanych odpowiedziach często zawarte są nagłówki, w których widnieje zbyt wiele wrażliwych danych, np. w postaci wersji używanego przez aplikację frameworku tak jak to ukazano na Rysunku 3.10.



```
x-powered-by: PHP/7.4.22 server: nginx/1.14.1
```

Rysunek 3.10 Przykład nadmiernej ekspozycji wrażliwych danych z <https://eportal.pwr.edu.pl/>

Dobrą praktyką jest, aby wyłączyć ww. nagłówki bądź też całkowicie pozbyć się ich z domyślnej konfiguracji. Domyślne komunikaty o błędzie oraz strony typu 404, dostarczane przez framework, również mogą nieść ze sobą wrażliwe informacje o lukach bezpieczeństwa w naszej aplikacji. Większość bibliotek jest *open-source* co oznacza, że każdy ma wgląd do ich zawartości oraz historii wszelkich poprzednich wersji. Więc, dla przykładu, jeśli w aplikacji, która odwiedzamy, wyskoczy strona z błędem o określonej zawartości, będziemy mogli porównać sobie tą zawartość z zawartością dostępną w repozytorium frameworku (aktualna, bądź któraś z poprzednich wersji) i w ten sposób domniemamy jego wersję używaną przez aplikację. Pozwoli to na sprawdzenie w bazie podatności na jakie dana aplikacja jest wrażliwa.

Skanowanie używanej przez serwer bazy danych czasem polega na analizie nagłówków, wysyłanych komunikatów o błędzie, do strony klienta. Jednak zazwyczaj nie są one wysyłane i najczęściej należy znaleźć alternatywną drogę do wykrycia używanej przez aplikację bazy. Jedną ze skutecznych, aczkolwiek wymagających większej wiedzy, technik jest analiza głównego klucza tabeli bądź dokumentu. Wiele popularnych oraz szeroko dostępnych baz danych wspiera mechanizm głównego klucza, który jest automatycznie generowany podczas tworzenia rekordu w bazie a następnie używa się go do bardzo

szynkowych wyszukiwań. Tak więc, znając algorytm, używany przez konkretną bazę do generowania ww. kluczy (często algorytmy te są przedstawiane w dokumentacjach), można porównać go do algorytmu szyfrowania głównego klucza używanego w analizowanej aplikacji. Aby go otrzymać, należy przeanalizować ruch *HTTP* podczas wysyłania zapytań na punkty końcowe. Przykład odpowiedzi od serwera zawierającego klucz główny w postaci „\_id” przedstawiono poniżej. [1]

```
{  
  "_id": "617fe9bfce793cc0f4da6103",  
  "username": "rafal",  
  "email": "rafal@gmail.com",  
  "phone": "123456789"  
}
```

Bardzo przydatną funkcjonalnością oferowaną przez *npm* (ang. *node package manager*) [11] jest wykonywanie audytu wszystkich zewnętrznych zależności aplikacji poprzez uruchomienie komendy *npm audit*. Generuje ona raport podatności naszej aplikacji oraz wskazuje która zależność powoduje powstanie danej luki bezpieczeństwa, co jest przedstawione na Rysunku 3.11.

```
Depends on vulnerable versions of snapdragon  
node_modules/micromatch  
  anymatch 2.0.0  
    Depends on vulnerable versions of micromatch  
      node_modules/watchpack-chokidar2/node_modules/anymatch  
      node_modules/webpack-dev-server/node_modules/anymatch  
        fast-glob ≤2.2.7  
          Depends on vulnerable versions of glob-parent  
          Depends on vulnerable versions of micromatch  
            node_modules/fast-glob  
              globby 8.0.0 – 9.2.0  
                Depends on vulnerable versions of fast-glob  
                  node_modules/globby  
                    fork-ts-checker-webpack-plugin 0.4.14 – 4.1.6  
                      Depends on vulnerable versions of micromatch  
                        node_modules/fork-ts-checker-webpack-plugin  
                        http-proxy-middleware 0.18.0 – 0.19.2  
                          Depends on vulnerable versions of micromatch  
                            node_modules/webpack-dev-server/node_modules/http-proxy-middleware  
                            readdirp 2.2.0 – 2.2.1  
                              Depends on vulnerable versions of micromatch  
                                node_modules/watchpack-chokidar2/node_modules/readdirp  
                                node_modules/webpack-dev-server/node_modules/readdirp  
                                nanomatch ≥0.1.1  
                                  Depends on vulnerable versions of snapdragon  
                                    node_modules/nanomatch  
                                    union-value *  
                                      Depends on vulnerable versions of set-value  
                                        node_modules/union-value  
  
50 vulnerabilities (19 moderate, 31 high)
```

Rysunek 3.11 Wynik audytu zewnętrznych zależności po wykonaniu komendy "npm audit"

Poprzez wykonanie komendy *npm audit fix* pozwoli na rozwiązywanie błędów dotyczących bezpieczeństwa poprzez zainstalowanie nowszych, prawdopodobnie poprawionych już wersji zależności.

### 3.5. Analiza architektury aplikacji

Dysponując wiedzą o architekturze praktycznie każdej współczesnej aplikacji internetowej można z pełną świadomością przeprowadzić jej audyt w celu określenia podatności na przeróżne ataki w sieci. W rozdziale zostały poruszone rozmaite sposoby oraz podejścia do uzyskiwania użytecznych informacji na temat aplikacji, które można na co dzień używać, a ponadto łączyć ze sobą w celu jeszcze większego ich dowartościowania. Każda osoba zajmująca się bezpieczeństwem powinna wiedzieć w jaki sposób nowoczesne aplikacje są budowane oraz powinna wskazać najlepsze praktyki do jej zabezpieczenia. Ponadto wiedza ta również pozwoliłaby na przeprowadzanie ataków w sposób samodzielny co umożliwiłoby znacznie szybsze wykrycie podatności – często przed atakującym z zewnątrz, co bez wątpliwości odbiłoby się negatywnie na wizerunku firmy odpowiedzialnej za dany produkt.

Często mówiąc o podatnościach aplikacjach internetowych pod uwagę brane są problemy leżące na poziomie kodu bądź występujące w wyniku niewłaściwie napisanego kodu. Te pierwsze można łatwo zauważać już wcześniej w architekturze aplikacji. Nieradko projekt architektoniczny aplikacji albo do wielu błędów związanych z bezpieczeństwem albo do stosunkowo małej liczby tych błędów w zależności od tego jak zaprojektowano i rozprowadzono zabezpieczenia aplikacji w całym kodzie. Z tego powodu umiejętność identyfikowania słabych punktów w architekturze aplikacji jest niezbędna. Funkcjonalności o słabych architekturach powinny być brane pod uwagę w pierwszej kolejności.

Architektura aplikacji często jest rozpatrywana na bardzo wysokim poziomie zamiast na niskim, gdzie tak naprawdę trzeba wykonać najwięcej pracy związanej z zabezpieczaniem. Może to być mylące dla kogoś, kto nie jest przyzwyczajony do rozpatrywania aplikacji z perspektywy projektowania.

Podczas badania aplikacji warto stworzyć sobie mapę jej architektury bezpieczeństwa. Opanowanie analizy architektonicznej nie tylko pozwoli na zmniejszenie wysiłku i niepotrzebnego nakładu pracy, ale też umożliwi wykrycie błędów w planowanych, przyszłych funkcjonalnościach poprzez wykrywanie wzorców, które powodowały pojawienie się błędów we wcześniejszych funkcjach.

## 4. Potencjalne zagrożenia aplikacji webowych

Po zaznajomieniu się z architekturą nowoczesnych aplikacji internetowych dysponujemy wiedzą, w których jej miejscach należy szukać luk bezpieczeństwa. Wiemy też jakie potencjalne podatności takowe luki wystawiają. Brakuje nam natomiast wiedzy w jaki sposób możemy je wykorzystać do przeprowadzenia pomyślnego oraz przemyślanego ataku na aplikację. W tym rozdziale skupię się na przedstawieniu jednych z najczęściej używanych i najbardziej niebezpiecznych ataków webowych oraz pokażę w jaki sposób można je przeprowadzić. Testy ataków będą przeprowadzane na własnej aplikacji – komunikatorze internetowym.

### 4.1. Cross-Site Scripting (XSS)

*Cross-Site Scripting* jest jedną z najczęściej występujących podatności webowych, który pojawił się jako odpowiedź na coraz to wzrastającą liczbę użytkowników Internetu. Atak ten polega na osadzeniu w treści strony własnego kodu *JavaScript*, co z kolei skutkuje możliwością wykonania niepożądanych akcji przez niczego nieświadomych użytkowników odwiedzających stronę czy aplikację.

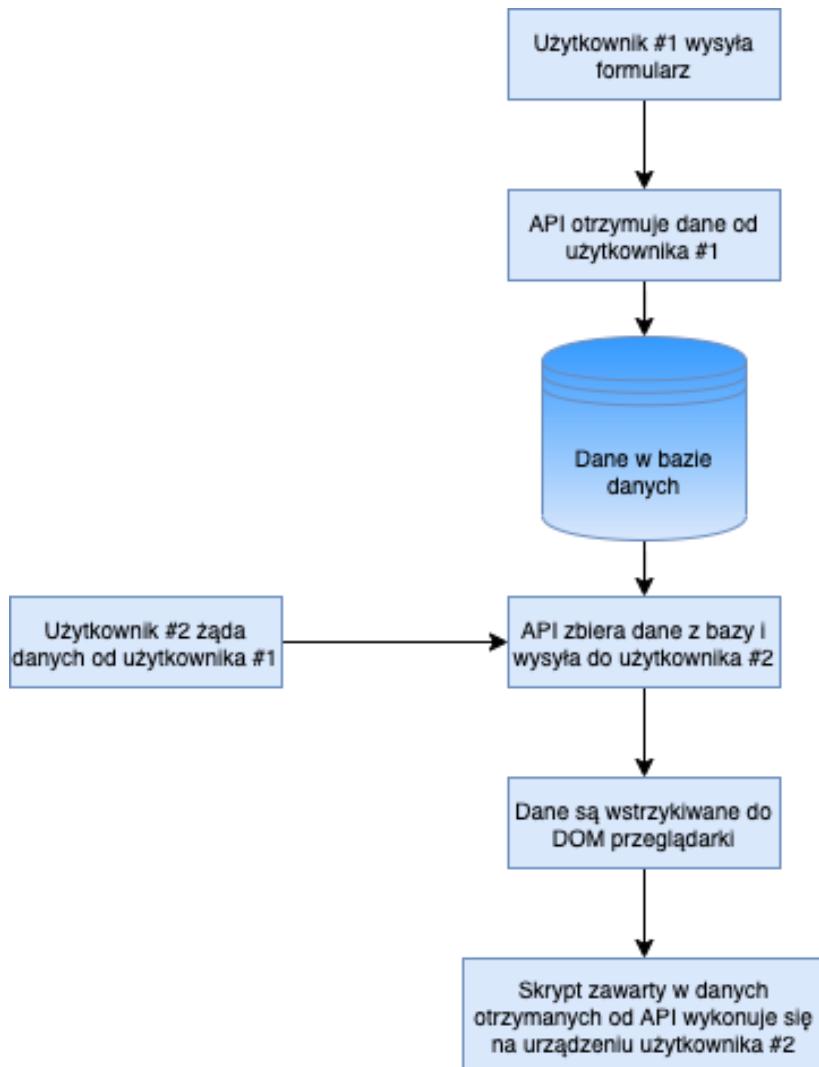
Ataki XSS możemy skategoryzować w wielu kontekstach, gdzie trzy najważniejsze z nich są następujące:

- *Stored* (kod jest umieszczany po stronie serwera – najbardziej złowroga odmiana)
- *Reflected* (kod przechowywany w linku, który następnie jest wysłany do ofiary)
- *DOM-based* (kod jest przechowywany oraz wykonywany w przeglądarce)

Poza wyżej wymienionymi, istnieje znacznie więcej kategorii ataków XSS, jednakże te trzy są głównymi i najczęściej występującymi w kontekście nowoczesnych aplikacji internetowych. Ponadto te trzy typy ataków zostały określone przez komitet *Open Web Application Project (OWASP)* jako najczęstsze wektory ataków XSS w sieci. Przed omówieniem każdej z trzech ww. kategorii ataków, warto w skrócie omówić jakie szkody może ze sobą ponieść pomyślnie przeprowadzony atak *Cross-Site Scripting*: [2]

- Wykonanie w przeglądarce skryptu, który nie został napisany przez właściciela aplikacji
- Atak (skrypt) może wykonywać się w tle będąc niezauważalnym oraz niewymagającym żadnej interakcji użytkownika
- Uzyskanie jakichkolwiek danych przechowywanych w danej aplikacji
- Możliwość wysyłania oraz otrzymywania danych ze złośliwego serwera WWW
- Każdy użytkownik może wprowadzić złośliwy skrypt, gdy dane przez niego wprowadzane nie będą kontrolowane oraz czyszczone
- Może zostać użyty do wykradnięcia tokenów autoryzacji i przejęcia sesji
- Możliwość edycji struktury oraz wyglądu strony co można skutecznie wykorzystać do przeprowadzenia ataku typu phishing

**Stored XSS**, czyli najbardziej szkodliwy a zarazem najpopularniejszy rodzaj ataku XSS, jest ku zaskoczeniu stosunkowo łatwy do wykrycia.



Rysunek 4.1 Schemat przykładowego przebiegu ataku typu Stored XSS

Schemat na Rysunku 4.1 przedstawia umieszczenie złośliwego skryptu przez użytkownika, który następnie jest przechowywany na bazie danych oraz pobierany przez innych użytkowników, którzy nieświadomie go wykonują na swoich maszynach. Tak umieszczony kod po stronie serwera oraz składowany na bazie danych, z którą serwer ten się łączy, może być widziany przez wielu innych użytkowników. Co więcej, jeśli takowy skrypt zainfekuje globalnie składowane dane, wszyscy użytkownicy aplikacji mogą zostać narażeni na atak XSS. [3]

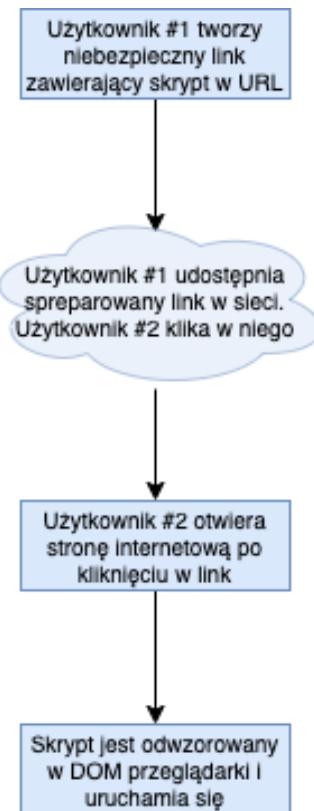
Dla przykładu, na stronie blogowej, jeśli atakujący umieści skrypt XSS w tytule blogu, to każdy użytkownik który zdecyduje się przeczytać dany blog, zostanie narażony na atak oraz nieświadomie wykona kod zaszyty w ww. tytule. Dlatego z tego powodu, atak typu *Stored XSS* jest bardzo szkodliwy dla jakiejkolwiek organizacji.

Jednak z drugiej strony, trwały charakter *Stored XSS* sprawia, że jest on stosunkowo łatwy do wykrycia. Dlatego regularne skany bazy danych w poszukiwaniu oznak skryptu XSS jest prostym, skutecznym oraz tanim rozwiązaniem łagodzącym skutki omawianego ataku. Jednakże nie zawsze rozwiązanie to się sprawdzi, gdyż skrypty niekoniecznie muszą być zapisane w formie zwykłego tekstu. Mogą to być dla przykładu ciągi binarne czy tekst zakodowany kodowaniem typu *base64*. Jest to jeden z kilku sposobów obejścia wykrycia skanerem bazy danych.

Wyżej poruszony przykład ze stroną blogową pokazuje nie tylko najczęściej spotykaną formę wstrzykiwania skryptu XSS przez atakujących, ale i też najłatwiejszą do zablokowania. Świadomy inżynier bezpieczeństwa czy programista będzie w stanie bezproblemowo powstrzymać próby ataku tego typu poprzez zbudowanie wyrażenia regularnego blokującego wstrzykiwanie jakichkolwiek skryptów przez użytkownika czy też budując regułę *CSP* (*Content-Security-Policy*) uniemożliwiającą wykonywanie skryptów wrzuconych bezpośrednio w strukturę aplikacji internetowej. [12]

Na koniec warto wspomnieć o tym, że atak XSS będzie zakwalifikowany jako typu *Stored XSS* wtedy i tylko wtedy, gdy zawartość tego ataku (np. skrypt) zostanie przechowyany na bazie danych. Poza tym, ten typ ataku nie posiada żadnych innych wymagań.

**Reflected XSS** jest stosunkowo prostą (z punktu programisty) odmianą ataku XSS. Polega na tym, że atakujący podsyła swojej ofierze link, w którym zaszyty jest złośliwy skrypt. Gdy atakujący wejdzie w dany link, wykona się skrypt zaszyty w linku a użytkownik, niczego nieświadomy, zostanie poszkodowany wykonanym kodem. *Reflected XSS* nie powinien być zapisywany na bazie danych, ani nawet być wysyłany często na serwer. Jedyne miejsce, w jakim powinien operować to klient przeglądarki. Schemat tego ataku został przedstawiony na Rysunku 4.2. [1]



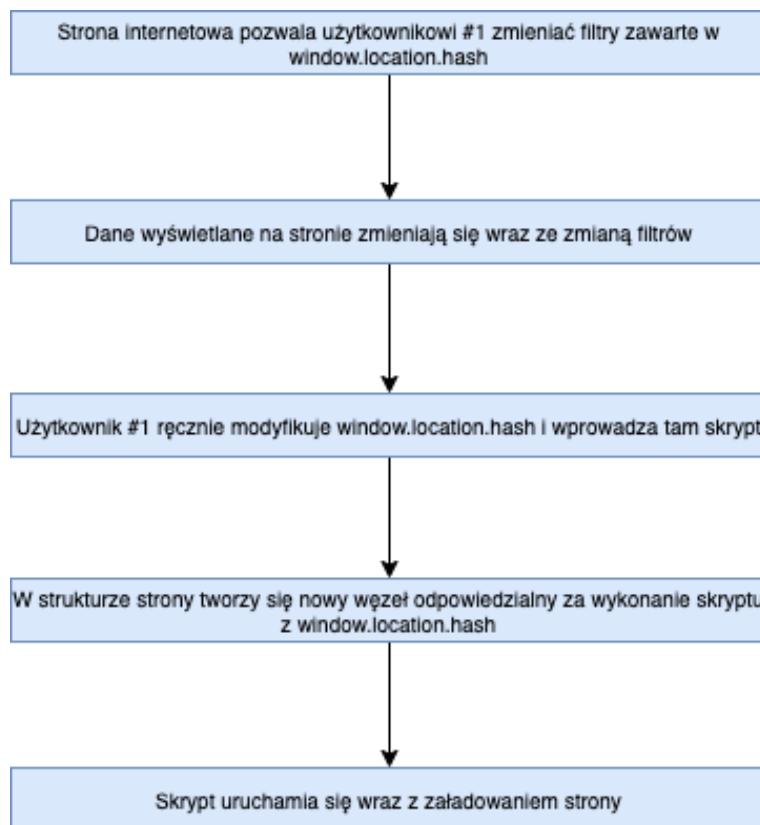
Rysunek 4.2 Schemat przykładowego przebiegu ataku typu Reflected XSS

W porównaniu do *Stored XSS*, ataki typu *Reflected XSS* są znacznie trudniejsze do wykrycia, gdyż nie są one przechowywane na bazie tylko bezpośrednio wymierzone w użytkownika. Jeśli użytkownik nie posiada żadnej wiedzy na temat bezpieczeństwa w sieci albo nie będzie szczególnie ostrożny podczas otwierania podszyłanych mu linków, może łatwo stać się ofiarą ataku. Co więcej, tego typu atak, może być osadzany w reklamach na stronie internetowej czy też emailach.

```
<a href="http://infected-site.com/execute?<script>alert('Zostałeś zhackowany!')</script>">Kliknij!</a>
```

Powyższy fragment kodu przedstawia atak *Reflected XSS* polegający na wstrzyknięciu kodu w URL, co jest stosunkowo proste dla atakującego do rozdystrybuowania go wśród docelowych użytkowników. Można śmiało powiedzieć, że *Reflected XSS* ukrywa się przed wykryciem znacznie lepiej niż *Stored XSS*, jednakże jego dystrybucja do większej ilości odbiorców jest bardziej trudna.

**DOM-based XSS** jest ostatnim z trzech głównych typów ataku XSS. Może on być albo typu *Stored* albo *Reflected*, a do prawidłowego działania potrzebuje jedynie klienta przeglądarki, źródła, w którym można obsadzić szkodliwy kod (najczęściej jest to element struktury strony posiadający możliwość przechowywania tekstu) oraz *DOM API*, którego zadaniem byłoby wykonanie wcześniej obsadzonego kodu w strukturze strony. *DOM-based XSS* nie wykonuje żadnych interakcji z serwerem przez co jest to prawie niemożliwe, aby wykryć go za pomocą jakichkolwiek narzędzi statycznych służących do analizy czy dostępnych, popularnych skanerów. Ponadto wykrycie tego ataku utrudnia konieczność posiadania dużej wiedzy o *DOM* przeglądarki oraz języku *JavaScript*, więc zazwyczaj tylko doświadczeni inżynierowi są w stanie jakkolwiek przeciwdziałać tego typu atakowi. Sprawę też komplikuje różnorodność implementacji *DOM* w dostępnych przeglądarkach, ponieważ może zaistnieć sytuacja, gdzie dana podatność będzie występować tylko w wybranych przeglądarkach, a w innych już nie. [1]



Rysunek 4.3 Schemat przykładowego przebiegu ataku typu *DOM-based XSS*

Rysunek 4.3 pokazuje typowy przebieg ataku typu *DOM-based XSS*. Podsumowując to w słowach, dla przykładu, użytkownik za pomocą znaku hash (#), bądź dodając parametry

zapytania (?=), może w linku strony zawrzeć dowolny skrypt, który następnie zostanie wykonany przez kod istniejący już w aplikacji. Na przykład, jeżeli aplikacja posiada mechanizm filtrowania wyników na podstawie parametrów zapytania w *URL*, to użytkownik będzie mógł statycznie wrzucić w link własny skrypt (zamiast prawidłowych parametrów zapytania pozwalających filtrować wyniki), a kod aplikacji, odpowiadający za filtrowanie wyników, spowoduje jego wykonanie.

```
example-site.com/?search=<script>alert(document.cookie);</script>
```

Tak zawarty parametr zapytania (*?search=*) najprawdopobniej będzie czytany przez aplikację za pomocą *window.location.search* – co staje się źródłem obsadzenia naszego kodu. Następnie, tak zczytany parametr, może być przekazany do wykonania dla niebezpiecznych funkcji JavaScript takich jak *eval* czy *innerHTML*, co spowoduje przeprowadzenie ataku typu *DOM-based XSS*. Ponadto cały ten atak nie wymaga żadnego zapytania do serwera, więc wszystko dzieje się w przeglądarce, co powoduje, że może on pozostać niewykryty przez długi okres.

Mimo że ataki XSS, we współczesnych aplikacjach internetowych, są mniej powszechnne niż w przeszłości, to nadal stanowią poważne zagrożenie dla firm oraz ich aplikacji. Jeśli programiści nie będą wystarczająco ostrożni przy implementacji funkcji związanych z umożliwieniem interakcji użytkownika z aplikacją, mogą otworzyć niebezpieczne luki bezpieczeństwa dla ataków typu XSS. Co więcej, dzisiejsze aplikacje webowe oferują bardzo wysoki poziom interakcji dla użytkownika co dodatkowo zwiększa powierzchnię dla omawianych w tym rozdziale ataków.

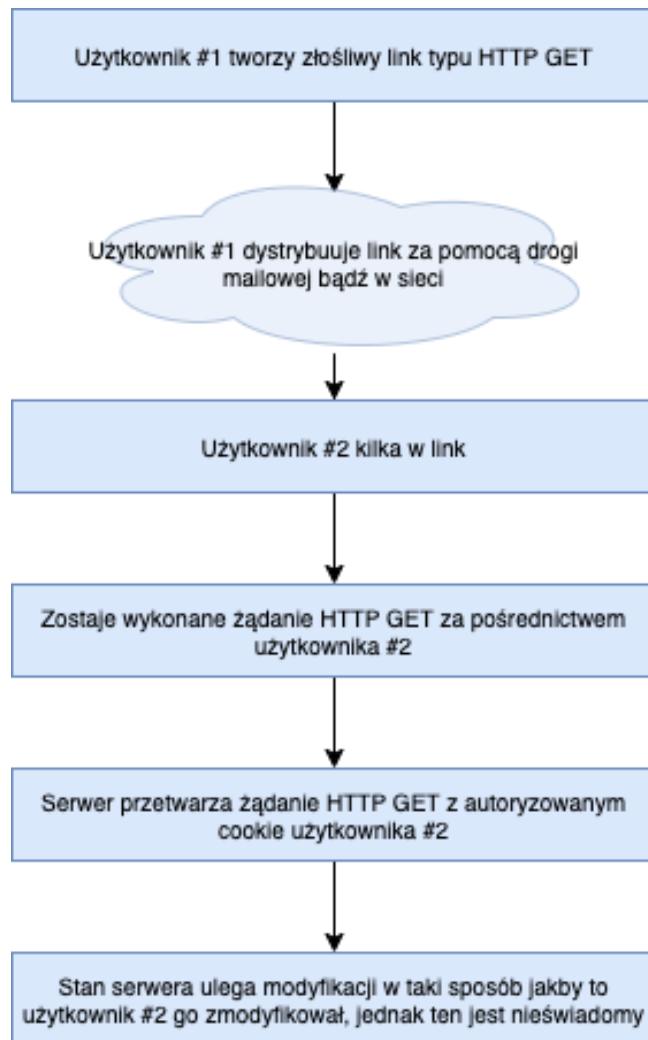
## 4.2. Cross-Site Request Forgery (CSRF)

Ataki *CSRF* do swojego działania wykorzystują sposób, w jaki operują przeglądarki oraz politykę zaufania między odwiedzaną witryną internetową a przeglądarką. Często do wykonania pewnych akcji w aplikacji potrzebujemy posiadać pewne przywileje, np. żeby zarządzać stroną, potrzebujemy mieć status administratora. W tym podrozdziale przedstawione zostaną sposoby na zaradzenie problemu nieposiadania określonych przywilejów, przy pomocy exploitów powodujących, że ww. administrator lub dowolne uprzywilejowane konto będą wykonywać określone operacje w naszym imieniu. Ponadto warto na wstępie dodać, że atak typu *CSRF* często zostaje niewykryty, gdyż wszelkie żądania przeglądarki dzieją się „za kulisami”, a więc nieświadomy użytkownik nawet nie będzie wiedział, że wykonuje pewne akcje w naszym imieniu oraz udostępnia nam zasoby.

Na początek warto wymienić najbardziej standardową odmianę ataku typu *CSRF*, czyli **Query Parameter Tampering**, co w wolnym tłumaczeniu oznacza manipulowanie parametrami zapytania, za pomocą hiperłącza. Najbardziej powszechną formą hiperłącza jest ta, która odpowiada żądaniom *HTTP GET*. Przykładowe żądanie tego typu jest przedstawione poniżej:

```
<a href="https://example-site.com"></a>
```

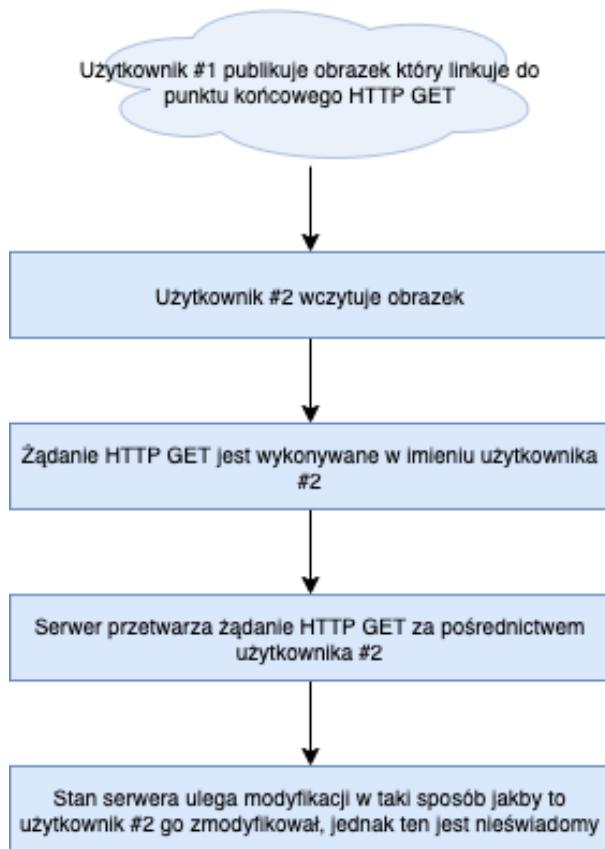
Kiedy serwer sieciowy otrzymuje tego typu żądanie, odpowiednie mechanizmy aplikacji, otrzymawszy odpowiednie parametry zapytania, je obsługują w celu zidentyfikowania użytkownika wysyłającego żądanie, identyfikacji przeglądarki, z której on zażądał oraz typu formatu danych oczekiwanych w zamian. [1]



Rys 4.4 Schemat przebiegu ataku CSRF poprzez delegację złośliwego linku do innego użytkownika

Rysunek 4.4 przedstawia w jaki sposób jest wykonywany atak typu Query Parameter Tampering. Potencjalny atakujący najpierw generuje złośliwy link a następnie dystrybuuje go do sieci poprzez email czy strony internetowe. Następnie nieświadomy użytkownik kliką w podesłany mu link, wysyłając przy tym żądanie typu *GET*, w imieniu atakującego. Końcowym efektem ww. ataku jest otrzymanie danych bądź wprowadzenie pewnych zmian na serwerze, przez osobę atakującą mimo, iż nie jest ona do tego typu działań uprawniona.

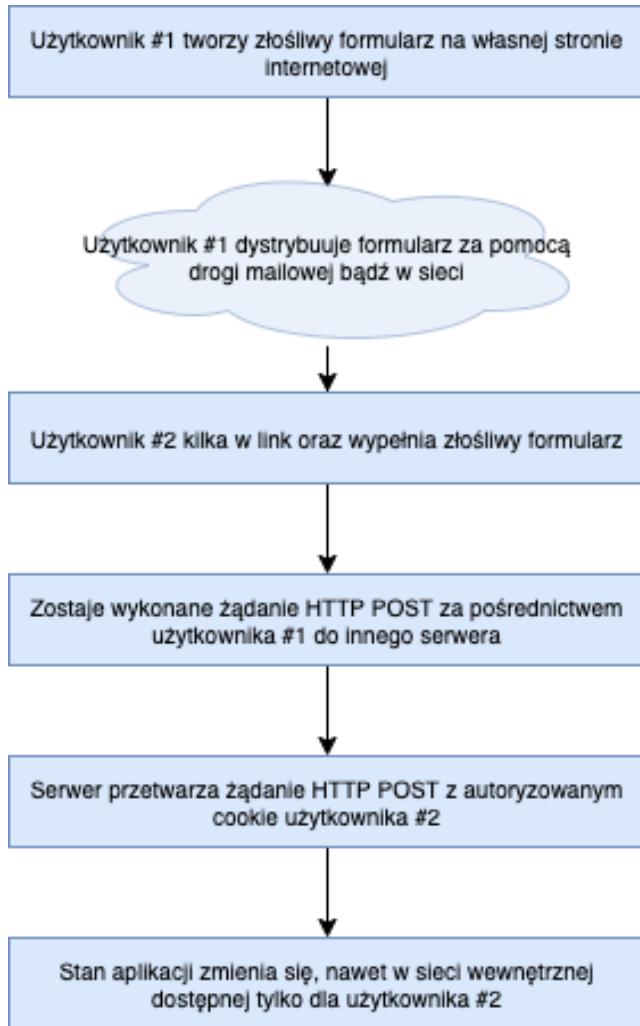
Większość tagów *HTML*, które przyjmują parametr *URL* (wykonanie zapytania do podanego serwera), domyślnie wykonuje żądania typu *GET*, przez co są one najbardziej narażone na ataki *CSRF*. W przykładzie wyżej, przedstawiony był przykład ataku *CSRF* polegający na wygenerowania złośliwego linku za pomocą hiperłącza. Jednakże link ten możemy również umieścić w źródle obrazków ładowanych na stronie w atakowanej przez nas aplikacji. Przewagą tego podejścia jest to, że wymieniany sposób nie wymaga żadnej interakcji ze stroną użytkownika (kliknięcie linku). Jedynie musi on wejść na stronę, gdzie znajduje się zainfekowany przez nas obrazek, a przeglądarka, próbując odczytać źródło obrazka do załadowania zawartości strony, automatycznie wykona żądanie *GET* do wskazanego przez nas (w *URL* obrazka) serwera. Przebieg opisanego działania jest przedstawiony na Rysunku 4.5.



Rys 4.5 Schemat przebiegu ataku CSRF poprzez umieszczenie złośliwego linku w obrazku

W związku z tym ważne jest, aby szukać dowolnego typu tagu *HTML*, który wysyła żądania *GET* do serwera, gdyż możemy ich śmiało użyć do przeprowadzenia ataku *CSRF*.

Warto również wymienić rodzaj ataku *CSRF* za pomocą żądań typu *POST*, *PUT* czy *DELETE*. Atak tego rodzaju wymaga jednak trochę większego nakładu pracy ze strony atakującego jak i wzmożonej interakcji użytkownika. Najczęściej tego typu ataki osadza się w formularzach, gdyż to właśnie one generują żądania ww. typu.



Rysunek 4.6 Schemat przebiegu ataku CSRF poprzez formularz wysyłający złośliwe żądanie

Na przedstawionym powyżej Rysunku 4.6 widzimy typowy przebieg ataku *CSRF* za pomocą żądania *POST*. Atakujący najpierw przygotowuje formularz, zawierający złośliwe żądanie, a następnie rozsyła go w sieci poprzez email. Co więcej, w *HTML*, może oznaczyć wybrane pola jako ukryte, więc użytkownik, wchodzący na zainfekowaną stronę, nawet nie będzie świadomy, że podjął przez przypadek dodatkowe, niechciane akcje zatwierdzając formularz.

```

<form action="https://example-site.com/money-transfer" method="POST">
    <input type="hidden" name="to_user" value="hacker">
    <input type="hidden" name="amount" value="10000">
    <input type="text" name="username" value="username">
    <input type="password" name="password" value="password">
    <input type="submit" value="Submit">
</form>

```

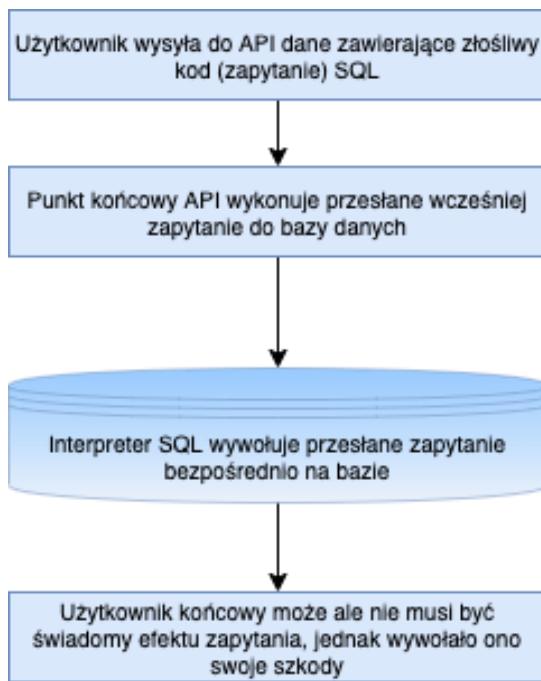
Następnie użytkownik, który wejdzie w podesłany mu link, wypełnia formularz oraz wysyła go w imieniu osoby atakującej. Analogicznie do poprzednich przykładów ataków *CSRF*, atakujący w ten sposób dostaje dostęp do zasobów aplikacji, za pośrednictwem swojej ofiary oraz może wykonywać różnego typu akcje, do których nie posiada praw jak

na przykład przelew pieniędzy na swoje konto z konta osoby poszkodowanej (w przypadku aplikacji bankowej).

Głównym powodem powstania ataków typu *CSRF* jest model zaufania opracowany przez komitety ds. standardów przeglądarek. Dopóki standardy te się nie zmienią, ataki *CSRF* będą w sieci powszechnie oraz często wykorzystywane. [1]

### 4.3. SQL Injection

Jednym z najbardziej powszechnych ataków na aplikacje internetowe jest *SQL Injection*. W zestawieniu *OWASP Top 10* z 2017 r. został on uplasowany nawet na pierwszym miejscu w kategorii najbardziej krytycznego zagrożenia bezpieczeństwa aplikacji internetowych. Ataki typu *SQL Injection* są kierowane w stronę *SQL-owych* baz danych pozwalając użytkownikowi zmodyfikować istniejące zapytania na serwerze, bądź też wprowadzać swoje własne zapytania. Skutkuje to nieautoryzowanym dostępem użytkownika do wrażliwych danych takie jak np. numery kard kredytowych, hasła czy adresy zamieszkania. Oprócz *SQL Injection* istnieją także inne podtypy ataku polegającego na „wstrzykiwaniu” kodu, tj. *Code Injection* czy *Command Injection*, o których można dowiedzieć się więcej w [1]. W niniejszym podrozdziale natomiast skupię się jedynie na omówieniu *SQL Injection*.



Rysunek 4.7 Schemat przebiegu ataku typu *SQL Injection*

Jak możemy zauważać na poglądowym schemacie ataku *SQL Injection* przedstawionym na Rysunku 4.7, użytkownik wpierw wysyła żądanego ze zmodyfikowanym zapytaniem do bazy – robi to za pomocą znaków modyfikacji czy ucieczki, jak np. „\”, tak aby spowodować, inną niż domyślona, interpretację sekwencji znaków. Następnie zmodyfikowane zapytanie zostaje wykonane na bazie powodując podjęcie akcji zdefiniowanych właśnie w tym zapytaniu. Użytkownik może, lecz nie musi, zaobserwować zmiany jakie spowodowało to zapytanie, jednakże atak sam w sobie się odbył i najprawdopodobniej wyrządził szkody na bazie danych.

Już po analizie samego przykładowego przebiegu ataku możemy się domyślić, jak niebezpieczny oraz szkodliwy dla organizacji może być atak tego typu. W większych

firmach czy korporacjach atak ten może wyrządzić szkody liczone w milionach, a ponadto narażone są zaufanie klientów oraz wizerunek firmy. Na szczęście w nowoczesnych aplikacjach webowych, zbudowanych w nowych technologiach, prawdopodobieństwo wystąpienia tego typu ataków znacznie się zmniejszyło w porównaniu do poprzedniej dekady [1]. Jednakże *SQL Injection* nadal jest wszechobecne w sieci, mimo zmniejszonej skali, dlatego warto pokazać je na przykładzie.

```
app.post("/users/user", function (req, res) {
  const userId = req.params.userId;
  await sql.connect("mssql://example:credentials@localhost/database");
  const result = await sql.query("SELECT * FROM users WHERE id = " + userId);

  return res.json(result);
});
```

Analizując powyższy kod źródłowy, możemy zauważyc, że przedstawiony na nim punkt końcowy wykonuje zapytanie do bazy danych w celu wyciągnięcia z niej użytkownika o id zadany w parametrze zapytania *URL*. Taka forma zapytania do bazy zakłada, że parametr *id* nie został w żaden sposób naruszony. Jednakże założenia takie bywają złudne, ponieważ dowolna osoba może wprowadzić dowolny parametr zapytania, przy czym zmieniając częściowo bądź całkowicie logikę zapytania do bazy.

Poniżej przedstawione są przykładowe trzy scenariusze wpływające na wyżej omawiane zapytanie do bazy oraz jego skutki.

- *userId* zamiast wartości liczbowej (123), dostałoby wyrażenie *I=I*. Spowodowałoby to wyciągnięcie z bazy wszystkich użytkowników, włącznie z ich danymi, gdyż *I=I* w *SQL* ewaluuje do wartości *true*, a więc zapytanie wyglądałoby w następujący sposób:

```
SELECT * FROM users WHERE id = true
```

- *userId* zamiast wartości liczbowej, dostałoby dodatkowe zapytanie, oddzielone średnikiem od *id*:

```
SELECT * FROM users WHERE id = 1; DROP TABLE users;
```

Skutkiem takowego zapytania jest usunięcie wszystkich wpisów o użytkownikach z bazy.

- zamiast samego *userId*, otrzymujemy zapytanie, w którym dla przykładu dodajemy do naszego konta większa ilość pieniędzy w aplikacji. Zapytanie wówczas wyglądałoby w następujący sposób:

```
SELECT * FROM users WHERE id = 1; UPDATE users SET money = 99999999 WHERE id = 1;
```

Gdy aplikacja zbyt mocno ufa danym wprowadzanym przez użytkowników, może skutkować to otworzeniu luki bezpieczeństwa dla ataków *SQL Injection*. Ataki te wymagają zrozumienia funkcjonalności atakowanej aplikacji, tak więc nie należą one do najłatwiejszych do skutecznego przeprowadzenia. *SQL Injection* jest bardzo niebezpiecznym, eleganckim oraz zdolnym do wykradnięcia wielu danych z konta, przejęcia go, podniesienia uprawnień czy po prostu idealnym do wywołania paniki atakiem.

#### 4.4. Denial of Service (DoS)

Ataku typu *DoS* posiada wiele swoich odmian - od rozproszonej wersji, czyli *DDoS*, obejmującej tysiące lub więcej skoordynowanych urządzeń oraz będącej jego najpopularniejszą formą, który polega na spowolnieniu serwera poprzez nieustanne zalewanie go żądaniami z dużej sieci urządzeń, aż do ataku *DoS* na poziomie kodu - *regex DoS*, który atakuje pojedynczego użytkownika spowalniając mu przykładowo walidację danych. Skutki ww. ataków są zróżnicowane i charakteryzują się m.in. spowolnionym działaniem serwera albo jego całkowitym wyłączeniem z użycia, podwyższenia rachunku za prąd wskutek nadmiernej pracy wykonywanej przez obciążone serwery czy blokowanie strumieniowania audio oraz wideo dla użytkowników końcowych. Wskutek tego, wykrycie ataków *DoS*, szczególnie tych mniej szkodliwych, jest nie lada wyczynem. „Zaletą” tego typu ataków jest to, że nie powodują one trwałych szkód w strukturze aplikacji, gdyż głównym ich celem jest znaczne pogorszenie wrażeń użytkowania aplikacji przez użytkowników. Dlatego warto jest się przyjrzeć typom ww. ataku z bliska.

**Regex DoS**, inaczej *ReDoS*, jest jedną z najpopularniejszych form ataku *DoS* w dzisiejszych aplikacjach webowych. Wyrażenia regularne, w aplikacjach, najczęściej służą do walidowania danych w formularzach oraz do wyszukiwania przeróżnych tekstów bądź też danych fraz w tekstuach.

```
const spacjeOrazAlfanumeryczne = /^[a-zA-Z\d\s]+$/i;
```

Wyrażenie regularne podane w przykładzie sprawdza, czy podany tekst zawiera jedynie znaki alfanumeryczne bądź spacje. Wyrażenie z przykładu może się okazać bardzo przydatne w walidowaniu na przykład nazwy użytkownika. *Regex*, z reguły, jest bardzo szybki, więc rzadkością jest sytuacja, w której spowalniałby on działanie serwera. Wyjątkiem są złośliwe wyrażenia regularne, które mogą stanowić zagrożenie w momencie, gdy pozwolimy, w naszej aplikacji, użytkownikom wprowadzać własne *regex*'y, np. poprzez formularze.

Dla celów demonstracyjnych, możemy sobie stworzyć złośliwe wyrażenie regularne, które będzie sprawdzać, czy w danym ciągu znaków kolejno występują zdefiniowane grupy (w poniższym przypadku będzie to grupa znaków *pwr*).

```
^(pwr)*+$
```

Te wyrażenie nazywamy złośliwym, ponieważ, gdy ciąg znaków nie podpasuje się do wyrażenia regularnego, silnik odpowiadający za analizowanie *regex*'u „cofnie” się do początku łańcucha znaków i zacznie na nowo szukać możliwych dopasowań. Problem ten można zilustrować przedstawiając testy przetwarzania ciągów znaków, gdzie w pierwszym przypadku będzie on spełniał kryteria wyrażenia regularnego, a w drugim nie.

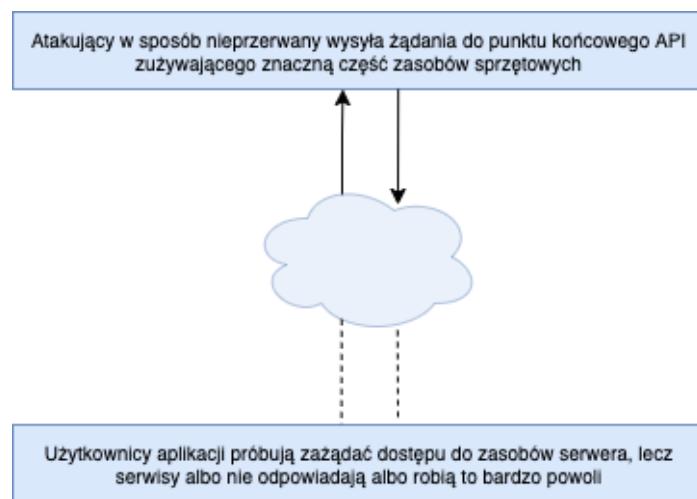
Rysunek 4.8 Test wyrażenia regularnego na prawidłowym ciągu znaków

Rysunek 4.9 Test wyrażenia regularnego na nieprawidłowym ciągu znaków

Porównując oba testy, przedstawione na Rysunkach 4.8 oraz 4.9, widzimy, że w pierwszym przypadku test, dla nawet tak długiego ciągu znaków, wykonał się bardzo szybko. Natomiast, gdy podamy tekst niespełniający kryteriów wcześniej zdefiniowanego złośliwego wyrażenia, czas wykonania testu drastycznie się zwiększa. Należy zwrócić uwagę, jak krótki jest łańcuch znaków, przy czym jego test zajął niemalże sekundę. Jest to też najdłuższy możliwy ciąg znaków, jaki mogłem wprowadzić w narzędziu, w którym przeprowadzałem testy, aby nie spowodować błędu wywołanym przekroczeniem limitem czasowym, który zapewnie wynosi 1 sekundę.

Z pomocą tego typu złośliwych wyrażeń regularnych możemy bez problemu obciążyc serwer aż do jego niezdolności bądź sprawić by strona klienta była bezużyteczna. Jedyną rzeczą jaką należy znaleźć jest odpowiednie miejsce w aplikacji (np. formularz) oraz przygotować odpowiednie dane na wejściu, które umożliwiłyby wykorzystanie ww. luki.

Innym przykładem ataku typu *DoS* są jego logiczne podatności. Atak ten polega na drenowaniu zasobów aplikacji przez nieautoryzowanego użytkownika. Schemat ataku znajduje się na Rysunku 4.10.

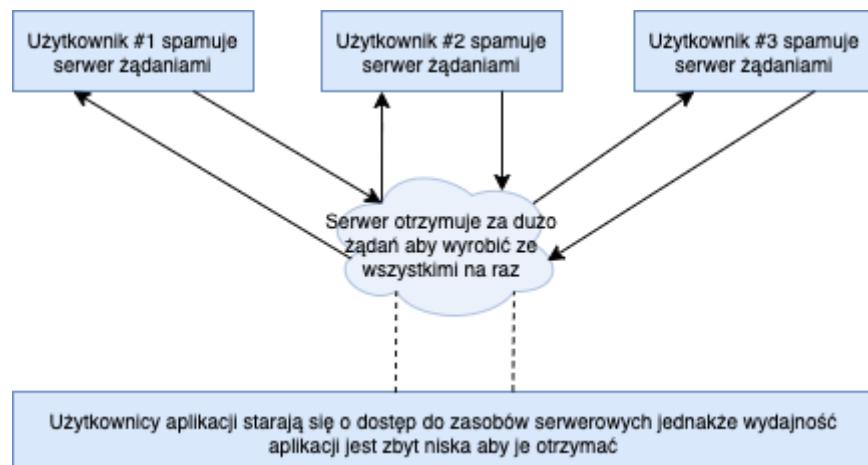


Rysunek 4.10 Schemat ataku wykorzystującego logiczne podatności na DoS

Ten typ ataku należy do jednych z najtrudniejszych do znalezienia. Wymaga on dużej wiedzy oraz doświadczenia w tym, gdzie i w jaki sposób go szukać. Rzeczą, którą na pewno trzeba wiedzieć, jest wiedza o tym, czym właściwie jest *DoS* i na czym polega.

Główne rzecz ujmując, *DoS* polega na wykorzystywaniu zasobów sprzętowych serwera lub też klienta w celu nieuzdatnienia ich do przeznaczonego użytku. Dlatego też, w pierwszej kolejności, należy sprawdzić miejsca w aplikacji pochłaniające dużo zasobów. Mogą to być na przykład zapytania do bazy, wysyłanie czy pobierania pliku lub też jakiekolwiek operacje na dysku. Za pomocą narzędzi developerskich w przeglądarce możemy zmierzyć czas poszczególnych żądań do serwera i stwierdzić, które z nich pochłaniają jak najwięcej zasobów oraz czasu – aby znaleźć idealne miejsce ataku. Gdy odnajdziemy już najbardziej podatny na atak *DoS* punkt końcowy, możemy zacząć „zalewanie” serwera zapytaniami (np. wrzucając co chwilę to samo zdjęcie do aplikacji). Jeśli programiści nie zabezpieczyli aplikacji poprzez ograniczenie zasobów per-żądanie, serwer, w trakcie ww. ataku *DoS*, z pewnością będzie działać wolniej albo i całkowicie może przestać działać, gdyż dostęp do zasobów może okazać się na daną chwilę niemożliwy. Oczywiście jest to tylko przykładowy przebieg ataku *DoS* wykorzystującego logiczne podatności aplikacji, ponieważ ataki te mogą się różnić ze względu na atakowane miejsca w aplikacji, tudzież atakowaną logikę.

Na koniec warto wspomnieć też o najpopularniejszej formie ataku *DoS* – *DDoS*, czyli *Distributed Denial of Service*. W przypadku tego rodzaju ataku, zasoby serwera są wyczerpywane przez bardzo dużą, skoordynowaną liczbę nieautoryzowanych użytkowników. Taka skala ataku ma znacznie większe powodzenie na zawieszenie aplikacji niż w przypadku zwykłego *DoS*. Ponadto celem tego typu ataków mogą być nawet bardzo duże firmy czy korporacje. W *DDoS* najczęściej uczestniczy kilku lub więcej osób atakujących, bądź wykorzystywane są też *botnet*y. Przykładowy przebieg ataku *DDoS* przedstawia Rysunek 4.11.



Rysunek 4.11 Schemat przebiegu ataku *DDoS*

*Botnet* to urządzenia przejęte przez hakera bądź grupę hakerów za pomocą złośliwego oprogramowania rozpowszechnianego w Internecie. Fakt ten powoduje, że wykrycie nieautoryzowanych klientów staje się uciążliwe. *Botnet*y najczęściej atakują warstwę sieciową zalewając serwer pakietami *UDP*, gdyż tam tak naprawdę najbardziej można wykorzystać ich potencjał. Nie są one natomiast zbyt skuteczne w ataku na warstwę aplikacji, np. *Regex DoS*, gdzie hipotetycznie można by było je wykorzystać w celu wysyłania ogromnych ilości danych podatnych na wyrażenia regularne.

Oprócz znacznego pogorszenia wydajności atakowanej aplikacji, ataki typu *DoS* mogą również powodować wyciek danych, dlatego warto też zwracać uwagę na logi oraz błędy

(np. w narzędziach developera), które pojawiają się w wyniku podejmowania wszelkich prób ataku *DoS*. [1]

#### 4.5. Luki bezpieczeństwa zależności zewnętrznych

Większość dzisiejszych aplikacji internetowych, w swoim kodzie, korzysta z zależności zewnętrznych. Nawet wielkie, komercyjne projekty korzystają z oprogramowania typu *open-source* budowanego przez wielu programistów z całego świata. Należą do nich między innymi *Reddit*, *YouTube*, *LinkedIn* czy *Twitch*. W Internecie istnieje aplikacja o nazwie *BuiltWith*, która analizuje jakich technologii używa podana aplikacja internetowa. Narzędzie to może okazać się bardzo pomocne w wyszukiwaniu wszelkich zależności danej aplikacji. Dla przykładu wykonałem analizę zależności *e-portalu* Politechniki Wrocławskiej i umieściłem ją pod danym linkiem: <https://builtwith.com/detailed/pwr.edu.pl>. Poleganie na zależnościach zewnętrznych jest na pewno bardzo wygodnym oraz przyspieszającym budowę aplikacji rozwiązaniem, jednakże potrafi też eksponować luki bezpieczeństwa, gdy jedna z używanych zależności takową posiada, które mogą być wykorzystane przez hakerów do ataku.

Jednym z głównych problemów wynikających z używania zależności zewnętrznych jest nieznajomość zawartości kodu tych zależności. Baza takiego kodu może być bardzo duża w zależności od rozmiarów biblioteki czy frameworki. Nie znając go, nie wiemy czy aby na pewno jest bezpieczny i nie wystawia jakiegokolwiek luki bezpieczeństwa. Ponadto analiza takiego kodu od początku do końca kosztowałaby firmę wiele czasu, a co za tym idzie – pieniądze, więc z reguły firmy nie godzą się na takie rozwiązania i godzą się na używanie zależności „na ślep”. Dodatkowo, zewnętrzne zależności są też na bieżąco aktualizowane, a zatem każda kolejna aktualizacja może przypadkowo uzewnętrznić nową podatność, co spowodowałoby konieczność ciągłego śledzenia zmian używanych bibliotek czy frameworków w celu analizy ich kodu, co z kolei kosztowałoby firmę kolejne dodatkowe pieniądze. Z tych właśnie powodów, zależności oraz integracje aplikacje są świetnym punktem startowym do przeprowadzenia ataku na aplikację. Po dokonaniu rozeznania jakich zależności używa dana aplikacja, możemy sprawdzić na jakie podatności dane zależności są wystawione oraz jak je możemy wykorzystać. Najpierw jednak warto wiedzieć w jaki sposób aplikacje integrują się ww. zależnościami.

Bardzo ważnym jest zrozumienie, w jaki sposób zbudowana jest integracja aplikacji z zewnętrznymi zależnościami, gdyż często dzięki temu można się dowiedzieć jaki jest typ danych w niej przesyłanych czy też jaki poziom uprawnień nadany jest przez główną aplikację dla zależności. Jednym ze sposobów integracji zewnętrznych zależności z aplikacją jest bezpośrednie wstrzyknięcie jej do kodu – ten sposób nazywa się scentralizowanym. Z kolei drugim sposobem – decentralizowanym, jest umieszczenie biblioteki czy frameworku na osobnym serwerze i konfiguracja *API* do jednokierunkowej komunikacji od głównej aplikacji do biblioteki. Oba rozwiązania mają swoje wady i zalety oraz stawiają wyzwania w podjęciu odpowiednich kroków do ich zabezpieczenia.

Wiele dzisiejszych bibliotek i frameworków jest obsługiwana w systemach kontroli wersji opartych na *Git*. *Git* jest dystrybuowanym narzędziem, co oznacza, że zamiast wprowadzać zmiany na scentralizowanym serwerze, każdy programista pobiera oddzielną kopię oprogramowania i lokalnie ją rozwija bądź wprowadza zmiany. Niestety wadą takiego rozwiązania jest możliwość pobrania niezweryfikowanej kopii oprogramowania z potencjalną luką bezpieczeństwa. Rozwiązaniem tego problemu są tzw. *Forks*. Posiadają one bardzo dobry poziom separacji, gdyż każdy *Fork* jest osobnym repozytorium, które ma swojego własnego właściciela oraz własne uprawnienia. *Forks* polegają na pobraniu ostatniej, zweryfikowanej wersji oprogramowania z repozytorium, co znacznie zmniejsza

ryzyko pobrania kodu z luką bezpieczeństwa bądź jakimkolwiek błędem. Niestety minusem jest fakt, że biblioteki czy frameworki nieustannie się rozwijają i aktualizowania swojego osobnego repozytorium do „oryginalnej”, najnowszej wersji często może okazać się bardzo żmudnym zadaniem zajmującym wiele czasu. [13]

Niebezpieczną oraz niepolecaną metodą integracji zewnętrznego oprogramowania są integracje „samo integrujących” się aplikacji. Jest to niezalecana metoda, gdyż z racji tego, że integracje te wykonują się automatycznie za pomocą predefiniowanego skryptu, nie wiemy co ten skrypt zawiera oraz pobiera do naszej głównej aplikacji. Trudno będzie potem namierzyć nam jakiekolwiek podatności oraz powody ich wynikania. Przykładem takiego typu integracji jest *WordPress*, który okrył się w Internecie bardzo złą sławą właśnie między innymi ze względu omawianych wyżej problemów.

Dla bibliotek małych rozmiarów (50-100 linijek kodu) idealną formą integracji będzie integracja bezpośrednio poprzez metodę kopiuj-wklej do kodu źródłowego. Jednakże integracja dużych zależności w przedstawiony wyżej sposób stanowi nie lada problem. Przykładowo programista może przypadkowo wrzucić do głównej aplikacji kod, który nie został jeszcze zweryfikowany przez co istnieje szansa, że w ten sposób wprowadzi lukę bezpieczeństwa w naszej aplikacji. Dodatkowo utrzymanie tak dużych zależności zintegrowanych w ww. sposób jest też problematyczne oraz czasochłonne.

Wiele dzisiejszych aplikacji, do integracji z zewnętrznymi zależnościami, używa pośredniej aplikacji nazywanej menedżerem paczek. Menedżer paczek to aplikacja zapewniająca prawidłowość w pobieranych zależnościach, pobierająca je z zaufanych źródeł oraz konfiguruująca je tak, aby można było z nich skorzystać bez względem na urządzenie, na którym uruchamiana jest aplikacja. Menedżery paczek upraszczają nam, dość już skomplikowane, metody integracji, zmniejszając rozmiary repozytorium oraz właściwie skonfigurowane, pobierają tylko te zależności, które są niezbędne do działania aplikacji. W bardzo dużych, biznesowych aplikacjach potrafi to zaoszczędzić wiele gigabajtów przesyłanych danych oraz wiele godzin komplikacji. Jednakże i to rozwiązanie nie pozostaje pozbawione wszelkich wad, gdyż w związku z pobieraniem i zarządzaniem tak wielka liczbą zależności, jest to prawie niemożliwe, aby odpowiednio ocenić i zweryfikować każdą zależność, co niesie ze sobą możliwość zainstalowania biblioteki z gotową luką bezpieczeństwa. [1]

Analizowanie każdej zależności oraz podatności, na jakie jest ona wystawiona, może nam zająć dość dużo czasu. Dlatego jednym ze sposobów na jego zaoszczędzenie jest znalezienie już znanych oraz jeszcze niezałatwanych luk bezpieczeństwa we frameworku czy bibliotece. Jednym z miejsc, w którym udokumentowane są ww. luki jest baza *Common Vulnerabilities and Exposures database (CVE)*. W bazie tej zawarte są opublikowane podatności najpopularniejszych bibliotek. Jest to duża zaleta dla hakerów, gdyż zawarte w *CVE* informacje bardzo ułatwiają znajdowanie oraz wykorzystywanie luk w zabezpieczeniach oraz ujawniają szczegółowe metody atakowania aplikacji. Jednakże, mimo tego udogodnienia jakie daje nam *CVE*, osoba atakująca nadal musi potrafić korzystać z technik rozpoznawczych, aby właściwie móc zidentyfikować zależności aplikacji oraz ich wersje i konfiguracje, a także być w stanie rozpoznać jakiej metody integracji zewnętrznych zależności używa dana aplikacja. [14][15]

## **5. Obrona przed atakami w sieci**

W poprzednim rozdziale wymieniłem jedne z najpopularniejszych oraz najniebezpieczniejszych ataków na aplikacje webowe, opisałem na czym one polegają i w jaki sposób się je przeprowadza. W niniejszym rozdziale skupię się na przedstawieniu sposobów ochrony przed wspomnianymi wyżej atakami oraz ich dokładnej analizie. Przedstawiona zostanie również koncepcja bezpiecznej architektury aplikacji internetowej, dzięki której nasza aplikacja domyślnie będzie posiadać szereg podstawowych zabezpieczeń, co znacznie zredukuje ilość jej podatności.

### **5.1. Bezpieczeństwo współczesnych aplikacji**

Współczesne aplikacje webowe są znacznie bardziej skomplikowane niż ich odpowiedniki sprzed kilku lat. Wiąże się to ze znacznym zwiększeniem obszaru ataku na takie aplikacje, między innymi przez większą interaktywność z użytkownikiem. Każdy inżynier bezpieczeństwa powinien dobrze znać obszary ataków na dzisiejsze aplikacje internetowe, gdyż wiedza ta znacznie mu ułatwi zabezpieczenie potencjalnych podatności oraz ukrycie architektury aplikacji przed hakerami. Aby dokładniej omówić kroki w budowaniu i utrzymywaniu bezpieczeństwa aplikacji, podzielię każdy na osobny podpunkt.

Pierwszym krokiem, w projektowaniu bezpiecznej aplikacji webowej, jest odpowiednie przygotowanie jej architektury oraz dogłębiańska analiza danych przekazywanych w aplikacji. W tej fazie (przed rozpoczęciem pisania aplikacji) znacznie łatwiej jest dostrzec oraz wyeliminować potencjalne luki bezpieczeństwa, ponieważ po wypuszczeniu produktu dla użytkowników końcowych możliwości zmian w architekturze, by wyeliminować daną podatność, są dość ograniczone.

W fazie pisania aplikacji bardzo ważnym krokiem jest wykonywanie dokładnych przeglądów kodu przed wpuszczeniem go na produkcję. Pozwoli to na wychwycenie typowych oraz częstych błędów programistów oraz uchroni aplikację przed wprowadzaniem do niej nowych podatności. Aby zachować jak najwyższą jakość kodu, powinien być on sprawdzany nie tylko przez kolegów z zespołu, ale i także przez niezależny zespół inżynierów bezpieczeństwa. Najważniejszymi punktami, które należy sprawdzić są:

- Sposób przepływu danych
- Sposób przechowywania danych
- Dane wysyłane do klienta (przedstawienie ich)
- Dana wysyłane na serwer (jakie akcje zachodzą na danych oraz sposób ich przechowywania)

Po zaprojektowaniu odpowiednie bezpiecznej architektury oraz sprawdzeniu kodu pod kątem bezpieczeństwa, kolejnym ważnym krokiem jest dokonanie analizy podatności potencjalnie występujących w wypuszczonej aplikacji pojawiających się między innymi w wyniku błędów. Wiele nowoczesnych aplikacji internetowych korzysta z takich form „skanowania” podatności jak programy „Big bounty”, wewnętrzne drużyny *Red* oraz *Blue*, testy penetracyjne czy nawet dodatkowe świadczenia firm dla inżynierów w celu większego przywiązywania uwagi do potencjalnych podatności. Podjęcie przynajmniej jednego z ww. działań dotyczących znajdowania podatności pozwoli firmie zaoszczędzić wiele pieniędzy, w wyniku potencjalnego ataku ze stosunkowo mniejszym nakładem finansowym czy też zapobiegnie utracie klientów.

Następnym krokiem, po wykryciu luk bezpieczeństwa, należy przeprowadzić dokładną ich analizę. Warto jest dokonać priorytetacji znalezionych podatności – przypisać każdej

poziom ryzyka, gdyż jedne mogą okazać się bardzo szkodliwe i znacznie wpływać na kondycję aplikacji, z kolei inne mogą wcale nie być tak groźne. Wówczas nie warto zajmować czasu dla programistów, który może się okazać bardzo cennym w przypadku naprawy niebezpiecznych podatności.

Gdy zostanie nadany priorytet już odkrytym lukom bezpieczeństwa, należy jak najszybciej przystąpić do naprawy najważniejszych z nich. Powinien zostać sporządzony zaplanowany oraz dokładny proces zarządzania znalezionymi podatnościami – pozwoli to na rejestrowanie postępów prac nad błędami oraz zapobiegnie przekroczeniu terminom. Ponadto, w czasie trwania napraw, warto zaimplementować logowanie akcji w miejscach występowania podatności, aby móc w odpowiednim czasie przeciwdziałać atakom przeprowadzanych w trakcie konserwacji aplikacji.

Po naprawieniu luk bezpieczeństwa, ważnym jest, aby napisać testy regresji zaimplementowanej poprawki. Testy potwierdzą, że zaimplementowana poprawka działa wystarczająco dobrze a luka bezpieczeństwa już nie istnieje.

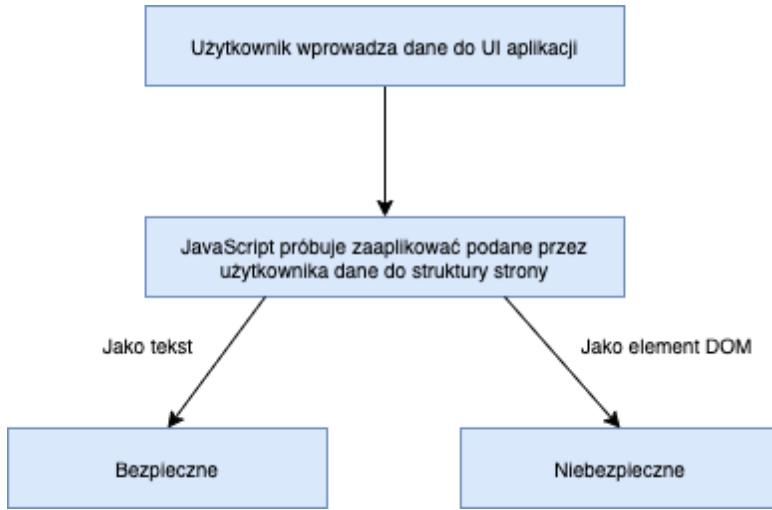
Na koniec warto pamiętać o aktywnym podejmowaniu wysiłków w celu ograniczenia ryzyka wystąpienia podatności w bazie kodu. Jest to tzw. strategia łagodzenia i jest to najlepsza praktyka, jeżeli zależy nam na bezpieczeństwu aplikacji, która ma swoje zastosowanie od samego projektowania architektury aż do fazy przeprowadzania testów zaimplementowanych poprawek. Strategia łagodzenia objawia się w dobrych praktykach bezpiecznego pisania aplikacji, jej bezpiecznej architektury, przeprowadzaniu testów regresji, bezpiecznym cyklu życia oprogramowania oraz „bezpiecznym” sposobie myślenia programisty podczas pisania kodu. Odniesienie się do każdego z wymienionych punktów, podczas projektowania oraz pisania aplikacji, znacznie zwiększy jej bezpieczeństwo, w związku z czym pozwoli zaoszczędzić dla firmy dużych ilości pieniędzy oraz zapobiegnie potencjalnej utracie klientów.

Więcej szczegółów na temat każdego z wyżej wymienionych punktów możemy znaleźć w książce [1].

## 5.2. Ochrona przed atakami XSS

W poprzednim rozdziale omówione zostały rodzaje ataków XSS oraz sposoby w jakie można czy też należy je przeprowadzać. Wiemy zatem jak wykorzystać podatność aplikacji na tego typu ataki zatem kolejnym krokiem jest nauka sposobów obrony przed nimi. Podrozdział ten skupi się na dobrych praktykach w zabezpieczeniu aplikacji internetowej przed wspomnianymi atakami.

Mimo że ataki XSS są jednymi z najgroźniejszych typów ataków na aplikację webową, to zabezpieczenie przed nimi jest stosunkowo proste. Dlatego też istotnym jest poznać najlepsze praktyki dotyczące obrony przed tymi atakami. Jedną z nich jest stosowanie się do zasady: „*Nigdy nie zezwalaj na przekazywanie danych, dostarczonych przez użytkownika, do DOM z wyjątkiem ciągów znaków*”. Jednakże istnieją aplikacje, w których jedną z funkcji jest modyfikowanie dokumentu *HTML* przez użytkownika. W takim przypadkach wiadome jest, że nie do końca możemy stosować się do powyższej zasady, gdyż pozbawilibyśmy wówczas główną funkcję aplikacji. W takiej sytuacji należy przynajmniej sprawdzać dane przesyłane przez użytkownika pod kątem zawartości niebezpiecznych skryptów czy linków oraz konwertować te dane na łańcuch znaków. Rysunek 5.1 przedstawia wyżej omawiany konspekt.



Rysunek 5.1 Typy przesyłanych danych a bezpieczeństwo

Aby upewnić się, że przesyłane, przez użytkownika, dane są na pewno ciągiem znaków należy dokonać sprawdzenia bądź konwersji typu danych w kodzie. Można to zrobić np. w poniższy sposób:

```
const isStringLike = payload => {
  try {
    return JSON.stringify(JSON.parse(payload)) === payload;
  } catch (error) {
    console.error(error);
  }
}
```

Powyższy kod, za każdym razem, próbuje przekonwertować otrzymane dane na obiekt z ciągiem znaków, a jeśli czynność ta się nie uda, akcja jest przerywana i następujący błąd zostaje wyświetlony w konsoli przeglądarki. W ten sposób nasza aplikacja jest chroniona przed niepożądanymi wstrzykiwanymi skryptami od użytkownika, gdyż metoda użyta w kodzie - `JSON.parse` - pozbywa się wszelkiego rodzaju funkcji (skryptów) z przesyłanych danych. Jeżeli nasza aplikacja posiada funkcję modyfikowania dokumentu strony poprzez interakcję użytkownika, warto też pamiętać, aby używać wbudowanej do tego metody `innerText`. Metoda ta posiada mechanizmy sprawdzania typu danych przesyłanych przez użytkownika i ewentualnego konwertowania ich na ciąg znaków. Jest to znacznie bezpieczniejsza opcja od jej odpowiednika `innerHTML`, który nie wykonuje takiego sprawdzenia, a zatem naraża aplikację na ataki typu *Stored XSS*.

```
document.body.innerText = payload;
```

Jednak czasami nie będziemy w stanie użyć dogodnych metod jak `innerText` wspomniane wyżej, np. wtedy, kiedy będziemy chcieli przepuszczać część *HTML-owych* tagów, a część, takich jak *script*, blokować. W takich przypadkach będzie trzeba zaimplementować własne mechanizmy walidowania danych, które będą sprawdzać czy przesyłane przez użytkownika dane nie zawierają potencjalnie niebezpiecznych (twarzyszących atakom) znaków. Możemy na przykład napisać własne wyrażenie regularne filtrujące, zdefiniowane wcześniej, niedozwolone przez nas znaki. Jednak przede wszystkim należy pamiętać, aby pod żadnym pozorem nie pozwalać użytkownikowi na wprowadzanie dowolnych danych dodawanych bezpośrednio do dokumentu strony internetowej.

Aby chronić się przed atakami typu *Reflected XSS*, należy również przeprowadzać sprawdzanie dynamicznie generowanych linków. Dla przykładu możemy przytoczyć sytuację, gdzie w aplikacji pozwalamy użytkownikom na tworzenie przycisków prowadzących do zdefiniowanych przez nich podstron. Jak wiemy, skoro link jest generowany dynamicznie, za pomocą wprowadzenia pewnych danych przez użytkownika to niewykluczone jest, że może zawierać on niebezpieczny skrypt. W tym przypadku możemy, na przykład, skorzystać z bardzo dobrze sprawdzonych i przetestowanych filtrów dla hiperłączy, które nowoczesne przeglądarki mają domyślnie wbudowane. Możemy je wykorzystać w następujący sposób:

```
const dynamicLink = "<script>alert('Zostałeś zhackowany!')</script>";

const dynamicRoute = () => {
  const hyperlink = document.createElement("a");
  hyperlink.href = dynamicLink;
  window.location.href = `https://example-page.com/${hyperlink.a}`;
}

dynamicRoute();
```

Tuż przed dynamicznym stworzeniem linku przez użytkownika, zostanie on przefiltrowany oraz poddany zakodowaniu do bezpiecznej formy linku – jest to domyślne zachowanie przeglądarki podczas tworzenia hiperłączy zapobiegające właśnie niepożądanym atakom XSS. Mechanizm przepuszczania danych, dostarczonych przez użytkownika podczas generowania linku, przez tagi hiperłączy - `<a></a>` - jest świetnym rozwiązaniem broniącym aplikacje przed atakami typu *Reflected XSS*, ponieważ oprócz „oczyszczania” linku z niedozwolonych znaków zapobiega również przechodzeniu do nieprawidłowych lub niewłaściwych adresów *URL*.

Innym środkiem zapobiegawczym jest kodowanie znaków *HTML* dostarczonych przez użytkownika (przy wysyłaniu danych). Kodowanie znaków, przedstawione na Rysunku 5.2, umożliwia określenie tylko tych, które mają być wyświetlane w przeglądarce, ale w taki sposób, aby nie można ich było interpretować jako *JavaScript*.

Character	Entity encoded
&	& + amp;
<	& + lt;
>	& + gt;
"	& + #034;
'	& + #039;

Rysunek 5.2 „Wielka piątka” kodowanych znaków *HTML*

Wykonując konwersje znaków przedstawionych wyżej w tabeli, nie ryzykujemy zmiany logiki wyświetlania elementów bądź danych w przeglądarce, a znacznie zmniejszamy ryzyko wykonania niepożądanego skryptu. Przedstawione wyżej znaki często są używane w skryptach *JavaScript*, tak więc kodując je, prawdopodobieństwo pojawienia się takowego skryptu w naszej aplikacji znacznie się zmniejsza.

W obronie przed atakami XSS znacznie może nam pomóc również *CSP* (*Content Security Policy*) do spraw dot. zapobiegania atakom XSS. *CSP* jest narzędziem wspieranym obecnie przez znaczną większość przeglądarek. Zawiera ono ustawienia, które pozwalają programiście zdefiniować reguły bezpieczeństwa kodu w aplikacji, a ponadto posiada ono także zabezpieczenia w postaci definiowania zewnętrznych skryptów, które mogą być ładowane do aplikacji, miejsc, w których mogą być ładowane oraz które *DOM API* mogą być używane w aplikacji.

Dużym ryzykiem, jakie ataki XSS niosą ze sobą, jest wykonanie skryptu pochodzącego z zewnętrznych źródeł. Wiadome jest, iż kod pisany samodzielnie jest tworzony z myślą o bezpieczeństwie użytkowników oraz stosowane są w nim najlepsze praktyki dot. bezpieczeństwa. Jednak nie można tego samego powiedzieć o kodzie pisany przez osoby trzecie. Jednym ze sposobów zmniejszenia ryzyka wykonywania skryptów zewnętrznych jest zredukowanie liczby dozwolonych źródeł skryptów. Możemy to osiągnąć poprzez zdefiniowanie w *CSP script-src* w następujący sposób:

```
Content-Security-Policy: script-src "self" https://example-subpage.com
```

Definicja *CSP* w powyższy sposób zapobiegnie ładowaniu się skryptów z innych domen niż domena, na której *CSP* jest ładowane (*self*) oraz <https://example-subpage.com>. Interpretacja *CSP* odbywa się również za pośrednictwem przeglądarki, tak więc jest to dodatkowe zabezpieczenie, ponieważ zestawy testów przeglądarek są bardzo obszerne.

Zdefiniowanie *script-src* w *Content Security Policy* zdecydowanie chroni aplikację przed zewnętrzny skryptami, aczkolwiek haker nadal może wstrzyknąć skrypt w zaufane serwery dokonując ataku XSS. *CSP* niestety nie dostarcza pełnej ochrony w omawianym zakresie, a jedynie zapewnia pewne środki zapobiegawcze przeciwdziałające najbardziej typowym scenariuszom ataków XSS (które programista może myłkowo przegapić). Dla przykładu, *Content Security Policy* domyślnie blokuje wykonywanie wbudowanych w dokument strony skryptów, czy też nie pozwala na wykonywanie się takich wbudowanych funkcji w *JavaScript* jak *eval*, która jest bardzo niebezpieczną oraz nierekomendowaną funkcją, gdyż konwertuje ona ciąg znaków na wykonywany kod, a zatem otwiera bramę dla ataków XSS. Jeśli jednak aplikacja wymaga tego typu funkcji do działania, warto napisać własną (i bezpieczną) wersję tejże funkcji, zamiast wyłączać domyślne zachowanie w *CSP*.

Jeśli chodzi o implementację *Content Security Policy*, jest ona stosunkowo prosta. Można to zrobić na dwa sposoby:

- Wysyłanie nagłówka *Content-Security-Policy* z każdym żądaniem do serwera – zawartością nagłówka powinna być konfiguracja *CSP*
- Umieszczenie meta tagu w pliku *HTML*. Tag ten powinien wyglądać w następujący sposób:

```
<meta http-equiv="Content-Security-Policy" content="script-src https://example-page.com;">
```

*CSP* jest zdecydowanie pierwszym krokiem, jaki świadomym programista bądź inżynier bezpieczeństwa powinien podjąć w celu przeciwdziałania atakom XSS. Jeśli wiadome jest, gdzie oraz jak będzie wykorzystywany kod aplikacji, można od razu przystąpić do napisania odpowiednich reguł *CSP*. Zdecydowaną zaletą jest to, że mogą być one dowolnie modyfikowane w późniejszych etapach rozwoju aplikacji. [1][2][3]

### 5.3. Ochrona przed atakami CSRF

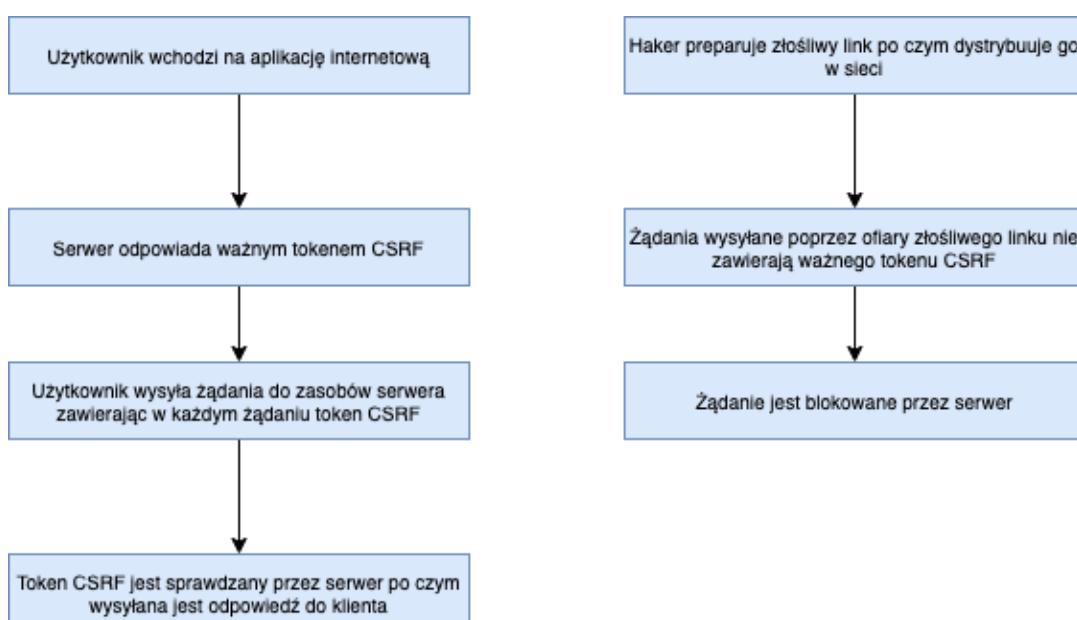
W poprzednim rozdziale omówiony został przebieg ataku *CSRF*, za pomocą którego, dla przypomnienia, możemy dokonywać nieautoryzowanych dla nas akcji przy pomocy autoryzowanego użytkownika. Pokazane zostały przykłady ataków *CSRF* za pośrednictwem hiperłączny, tagów `<img></img>` oraz poprzez wysyłane formularze przy użyciu metody *HTTP - POST*. Dowiedzieliśmy się jak ataki *CSRF* potrafią być niebezpieczne, a ponadto ciężkie do wykrycia przez użytkownika. W tym podrozdziale nauczymy się jak skutecznie bronić się przed tego typu atakami.

Na początku warto pokazać jeden ze sposobów obrony przed atakami *CSRF* dystrybuowanych za pomocą hiperłączny. Ponieważ źródło żądań przy atakach *CSRF* zazwyczaj różni się od źródła atakowanej aplikacji, możemy załagodzić tym atakom poprzez sprawdzanie ich pochodzenia i porównywania go z dozwolonymi przez nas wiarygodnymi źródłami.

W przypadku protokołu *HTTP* istnieją dwa nagłówki, które wskazują na pochodzenie żądania do aplikacji. Są to odpowiednio *Origin* oraz *Referer*. Różnica między nimi jest taka, że pierwszy z nich występuje nie tylko w żądaniach *HTTP*, ale i również *https*, a drugi może zostać „wyłączony” poprzez dodanie atrybutu *rel=noreferrer* do linku.

Walidację oraz sprawdzenie występowania nagłówków powinniśmy sprawdzać w kodzie po stronie serwera. Jeśli jest to możliwe, powinniśmy sprawdzać czy w żądaniu występują oba nagłówki. Jeżeli żaden nie jest obecny przy żądaniu, możemy z góry założyć, że żądanie to nie jest standardowe i powinno zostać odrzucone. Jednak, niestety, rozwiązanie to nie jest wystarczające, ponieważ, gdy haker zdobędzie dozwolone źródła żądań w danej aplikacji, będzie mógł tworzyć ataki *CSRF* udające, że pochodzą właśnie z tych źródeł. Dlatego sprawdzanie nagłówków jest jedynie podstawową oraz pierwszorzędną formą ochrony przed atakami *CSRF* i powinna być ona stosowana razem z innymi sposobami obrony.

Jedną z nich są tokeny *CSRF*. Wiele dzisiejszych aplikacji korzysta z tej formy obrony, ponieważ jest ona bardzo skuteczna, a jej implementacja nie należy do skomplikowanej.



Rysunek. 5.3 Schemat działania tokenów CSRF

Jak widzimy na Rysunku 5.3, przebieg działania tokenów *CSRF* nie jest wcale skomplikowany. Przebieg obrony bazującej na tokenach *CSRF* wygląda tak:

1. Po wysłaniu żądania do serwera przez klienta, serwer generuje podpisany kryptograficznie token *CSRF* za pomocą algorytmu z bardzo niskim współczynnikiem kolizji. Gwarantuje to unikalność wygenerowanego tokenu. Następnie token zwracany jest do klienta.
2. Token jest wysyłany przy każdym żądaniu klienta na serwer, gdzie jest walidowany, czy aby na pewno jest on autentyczny i nie wygasł. Jeśli walidacja nie przechodzi pomyślnie, żądanie jest natychmiastowo odrzucane.
3. W związku z tym, że każde żądanie na serwer wymaga ważnego tokenu *CSRF*, atakującemu bardzo trudno jest skonstruować atak na aplikację. Po pierwsze, będzie musiał on w jakiś sposób zdobyć ważny token, a po drugie – atak będzie mógł odbywać się tylko wobec jednego użytkownika za jednym zamachem, gdyż każdy token jest unikalny i przypisany tylko do jednej osoby. Co więcej, tokeny *CSRF* posiadają swoją datę ważności, więc bardzo prawdopodobnym jest wygaśnięcie tokenu w pierwszej kolejności zanim użytkownik zdąży otworzyć zainfekowany link.

Podany wyżej sposób zapobiegania atakom *CSRF* należy do architektury *Stateful*. Jednakże większość nowoczesnych aplikacji internetowych jest projektowana w architekturze *Stateless*. Niezbyt mądrym posunięciem byłaby zmiana architektury aplikacji tylko po to, aby zaimplementować wyżej przedstawiony sposób ochrony. Na szczęście tokeny *CSRF* mogą dość łatwo zostać zaimplementowane nawet w architekturze *Stateless*, ale będzie trzeba je ponadto odpowiednio zaszyfrować, aby zapewnić poufność danych. Taki token *CSRF* powinien zawierać unikalny identyfikator użytkownika, do którego należy token, datę ważności oraz zaszyfrowaną wartość, której klucz istnieje tylko na serwerze. Łącząc te wszystkie elementy w całość, otrzymujemy token *CSRF*, który jest nie tylko praktyczny, ale i również zużywa znacznie mniej zasobów niż jego alternatywa dla architektury *Stateful*.

Ponieważ najczęstsze oraz najłatwiej rozpowszechniane ataki *CSRF* biorą się z żądań *HTTP* typu *GET*, ważnym jest by odpowiednio zaprojektować *API* w taki sposób, aby załagodziło ryzyko ataków ww. typu. Żądania *HTTP* typu *GET* nie powinny przechowywać ani modyfikować danych po stronie serwera, ponieważ, w przeciwnym wypadku, utworzyłoby to poważną lukę bezpieczeństwa dla ataków *CSRF*.

```
const getAndUpdateUser = async (req, res) => {
  try {
    const user = await getUserById(req.query.id);

    if (req.query.doUpdate) {
      user.update(req.doUpdate);
    }

    return res.json(user);
  } catch (error) {
    console.error(error);
  }
};
```

Kod źródłowy przedstawiony powyżej pokazuje punkt końcowy *API* - szczególnie podany na atak *CSRF*. Funkcja pobiera użytkownika z bazy a następnie modyfikuje go,

czego nie powinno się robić dla żądań *HTTP* typu *GET*, ponieważ pozwala to w łatwy sposób zaatakować dany punkt końcowy za pomocą odpowiednio spreparowanego linku. Dlatego też rekomendowane jest, aby nie używać żądań *GET* do wykonywania jakichkolwiek operacji modyfikujących dane na serwerze. [1][16]

#### 5.4. Ochrona przed SQL Injection

*SQL Injection* jest jedną z najbardziej powszechnych odmian ataków polegających na wstrzykiwaniu złośliwego kodu. W związku ze swoją dużą popularnością (szczególnie w okresie ostatniej dekady) powstało wiele rozwiązań oraz dobrych praktyk chroniących przed atakami tego rodzaju, a więc implementacja obrony przed *SQL Injection* jest dzisiaj stosunkowo proste, jednakże programiści mimo wszystko i tak muszą uważać podczas projektowania interfejsów służących do komunikacji z bazą danych typu *SQL*.

Wykrycie wyżej omawianego ataku jest łatwym zadaniem, gdyż ataki te zazwyczaj występują w interpretatorach *SQL*. Odpowiednie strategie wykrywania tych ataków oraz łagodzenia ich skutków w znacznej mierze ograniczą podatność aplikacji internetowej na *SQL Injection*. Przed przystąpieniem do próby wykrycia występowania tego ataku, warto zaznajomić się z różnymi jego odmianami a następnie znaleźć miejsce w kodzie aplikacji, gdzie atak ten może nastąpić – będą to najprawdopodobniej punkty końcowe naszego *API* służące do pobierania/zapisywania danych do bazy, a więc strona serwera.

Jednym ze sposobów zabezpieczania się przed atakiem *SQL Injection* są spreparowane zapytania. Zmniejszają one znacznie ryzyko wystąpienia tego ataku za pośrednictwem danych przygotowanych przez użytkownika, są bardzo łatwe do nauczenia oraz w implementacji i znacznie ułatwiają debugowanie zapytań *SQL*. Działają one na zasadzie wpierw komplikacji zapytania *SQL* z wartościami zastępczymi dla zmiennych, a następnie podmianie tych zastępczych wartości na zmienne zdefiniowane przez programistę. W wyniku tego dwuetapowego procesu, intencja zapytania *SQL* jest ustalana już przed dostarczeniem do niej jakichkolwiek danych od użytkownika. Bez tego, użytkownik mógłby całkowicie zmienić intencje zapytania, np. z *SELECT \* FROM tbl\_users* na *DELETE \* FROM tbl\_users*, co spowodowałoby usunięcie wszystkich wpisów z tabeli użytkowników zamiast ich odczytu. Spreparowane zapytania eliminują większość podatności związanych z *SQL Injection* a ponadto są wspierane przez większość poważnych baz danych *SQL*. Jedyną znaczącą wadą korzystania ze spreparowanych zapytań jest nieznacznie obniżenie wydajności aplikacji.

```
PREPARE query FROM 'SELECT * FROM users WHERE age >= ?';
SET @age = 18;
EXECUTE query USING @age;
DEALLOCATE PREPARE query;
```

Powyżej przedstawiony jest przykład spreparowanego zapytania w bazie danych *MySQL*. Najpierw, za pomocą *PREPARE*, przygotowujemy zapytanie do komplikacji, potem przypisujemy wartość do zmiennej *age*, a następnie wykonujemy wyżej spreparowane zapytanie za pomocą *EXECUTE* przypisując zmienną *age* do wartości zastępczej w zapytaniu (znak zapytania). Na koniec, za pomocą *DEALLOCATE*, uwalniamy z pamięci przypisanie zapytania po to, aby użyć jej w przyszłości w innym kontekście. Napisany powyżej kod *SQL* uchroni aplikację przed wykonywaniem dodatkowych, niechcianych zapytań dostarczanych przez atakującego, dlatego tak ważne jest, aby implementować spreparowane zapytania, gdzie tylko występuje ryzyko ataku *SQL Injection*.

Oprócz uniwersalnych sposobów ochrony przeciwko *SQL Injection*, konkretni dostawcy baz danych zaimplementowali specyficzne dla danej bazy metody ochrony przez ww.

atakami. Na przykład, w *MySQL*, do dyspozycji mamy funkcję *QUOTE*, która powoduje usunięcie z zapytania wszelkich ukośników wstecznych, pojedynczych apostrofów czy też wartości *NULL*. Funkcja ta zwraca poprawnie ujęty, zwalidowany oraz w pojedynczych apostrofach ciąg znaków.

```
SELECT QUOTE('test"zapytania');
```

Używanie specyficznych dla danej bazy danych środków ochrony przed *SQL Injection* znacząco zredukuje ryzyko wystąpienia tego ataku poprzez, między innymi, tak jak w wyżej przedstawionym przykładzie, filtrację niedozwolonych znaków w zapytaniu. Jednakże ten sposób ochrony nie powinien być uznawany jako kompleksowa ochrona przed ww. atakiem, a jako pewna metoda łagodząca jego skutki.

Podczas zabezpieczania aplikacji internetowej przed atakami *SQL Injection*, należy również wziąć pod uwagę inne rodzaje ataków polegających na wstrzykiwaniu kodu. Mogą to być np. *Code Injection* czy *Command Injection*. Informacje na temat zabezpieczenia aplikacji przed tego rodzaju atakami można znaleźć tutaj [1]. Generalnie mówiąc, powinniśmy szukać miejsc, w kodzie źródłowym, w których może istnieć podatność zewnętrznego wstrzyknięcia kodu oraz stosować najlepsze, domyślnie bezpieczne praktyki programowania aplikacji, aby jak najbardziej zmniejszyć ryzyko wystąpienia jakichkolwiek podatności na ataki typu *Injection*. [17]

## 5.5. Przeciwdziałanie DoS

Ataki *DoS* zazwyczaj polegają na wyczerpaniu dostępnych zasobów systemowych przez co może być je ciężko wykryć, gdy nasz serwer nie ma zaimplementowanych odpowiednich mechanizmów logowania zdarzeń. Dlatego pierwszą czynnością, jaką należy wykonać, aby zapobiec tego typu atakom jest zaimplementowanie wyczerpującego mechanizmu logowania żądań. System ten powinien sprawdzać jakie żądania są w danej chwili wykonywane oraz czas ich odpowiedzi. Ponadto warto też zaimplementować miarę wydajności wszystkich asynchronicznych akcji w naszej aplikacji – pozwoli to nam na wykrycie potencjalnych podatności na ataki *DoS*, a oprócz tego zaoszczędzi to wiele czasu, gdyż późniejsze namierzenie takiej podatności manualnie może okazać się nie lada wyzwaniem. Podczas budowania strategii łagodzenia atakom typu *DoS*, musimy także wziąć pod uwagę wszystkie trzy płaszczyzny tego typu ataków. Wśród nich są to wyczerpywanie zasobów serwera, wyczerpywanie zasobów klienta oraz drenowanie zasobów aplikacji.

Na początku warto wziąć pod uwagę, najłatwiejszy w zapobieganiu, atak typu *Regex DoS*. Już w samej fazie sprawdzania kodu możemy wykluczyć potencjalne podatności na ww. atak. Wystarczy szukać w wyrażeniach regularnych znaków „+” oraz „\*”, które odpowiednio oznaczają znalezienie wszystkich potencjalnych ciągów znaków oraz nieskończoną ilość razy. Można do tego użyć skanera wyrażeń regularnych, aby znalazł właśnie tego *Regex*'y. Kolejnym krokiem jest zwracanie uwagi na miejsca w aplikacji, które są zdolne do wykonania dostarczanych przez jej użytkownika wyrażeń regularnych. Należy, w miarę możliwości, eliminować takie miejsca, gdyż nie możemy do końca ufać użytkownikom oraz danym jakie wprowadzają. Zatem należy eliminować miejsca w których mogą wystąpić ww. podatności oraz warto też sprawdzić czy zewnętrzne aplikacje, z którymi się integrujemy, nie otwierają wspomnianych luk bezpieczeństwa.

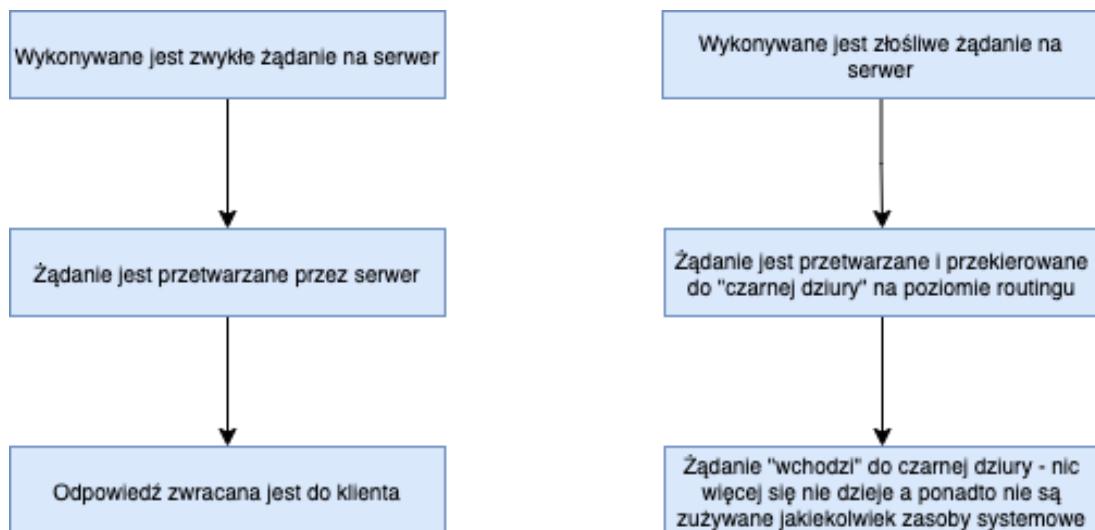
Logiczny *DoS* z kolei jest znacznie trudniejszy w zabezpieczeniu. Atak tego typu zazwyczaj jest powodowany nie do końca przemyślaną logiką aplikacji przez programistów. Bowiem to oni mogą przypadkowo „otworzyć” punkty końcowe podatne na wyczerpanie zasobów. Trudno jest też skategoryzować podatność aplikacji na tego typu atak, ponieważ

nie jest to atak binarny a mierzony na dużą skalę. W dużej mierze atak ten zależy również od zasobów systemowych zarówno atakującego jak i ofiary, ponieważ gdy atakujący będzie w posiadaniu dużej ilości zasobów to będzie w stanie i tak zaszkodzić nawet dla bardzo dobrze napisanej oraz wydajnej aplikacji. Z kolei ofiara posiadająca mocny komputer nie odczuje aż tak skutków tego ataku jak ktoś ze starszym sprzętem. Dlatego też w celu zabezpieczenia przed tego typu atakami, należy szukać miejsc w aplikacji, w których wykorzystywane są krytyczne zasoby systemu.

Zapobieganie atakom *DDoS* należy do bardzo trudnych. Podczas gdy celem ataków *DoS* jest zazwyczaj konkretne miejsce, bug w aplikacji, ataki *DDoS* są znacznie prostsze. Najczęściej przebiegają one z kilku bądź więcej skoordynowanych ze sobą źródeł, a ich głównym celem jest zasypanie aplikacji dużą ilością legalnie wyglądających żądań. Przez to normalni użytkownicy aplikacji są od niej całkowicie odcięci albo jej działanie jest znacznie spowolnione. Ataków *DDoS* nie da się powstrzymać, jednakże można im załagodzić na kilka sposobów.

Najłatwiejszym sposobem na obronę przed atakami *DDoS* jest inwestycja w system zarządzania pasmem. System ten analizuje każdy pakiet wchodzący do aplikacji, skanuje go pod kątem złośliwości i następnie decyduje, czy go przepuszczać dalej czy nie. System ten jest bardzo efektywny, ponieważ jest w stanie przechwytywać duże ilości żądań sieciowych, podczas gdy infrastruktura naszej aplikacji nie byłaby w stanie tego zrobić.

Innym sposobem przeciwdziałania atakom *DDoS* jest tzw. *blackholing*. Polega on na dołożeniu, do naszego głównego serwera aplikacji, innych serwerów pełniących funkcję swego rodzaju „kontenera” na podejrzanie wyglądające żądania. Przykład działania tego mechanizmu ukazany jest na Rysunku 5.4.



Rysunek 5.4 Schemat działania systemu *blackholing*

Podejrzane pakiety kierowane są od razu do serwera *blackhole*, który udaje główny serwer aplikacji, lecz nie podejmuje on żadnych działań po otrzymaniu żądania. Wadą tego rozwiązania jest to, że dobre pakiety również mogą zostać odfiltrowane do serwera *blackhole* jeśli nie wpasują się w określony wzorzec. Technika ta sprawdzi się w przypadku ataków *DDoS* na mniejszą skalę, lecz w przypadku dużych może sobie nie poradzić.

W obu technikach przeciwdziałania atakom *DDoS*, przedstawionych powyżej, wykorzystywane jest swego rodzaju odfiltrowywanie pakietów. W związku z tym należy pamiętać, że zbyt rygorystyczne filtry oraz wzorce mogą również odfiltrowywać pożądany ruch, dlatego też przed zaimplementowaniem którejś z powyższych metod przeciwdziałania

*DDoS* warto przeanalizować jaki ruch generują użytkownicy naszej aplikacji i dopiero na tej podstawie dopasować odpowiednie filtry. [1]

## 5.6. Zabezpieczanie zewnętrznych zależności

W tym podrozdziale główna uwagę skupię na sposobach ochrony aplikacji przed podatnościami, które mogą powstać podczas integracji z zewnętrznymi zależnościami.

Pierwszym sposobem jest zestawienie oraz analiza „drzewa” wszystkich zależności, paczek, bibliotek itd., które integrujemy z naszą aplikacją. Wiele z nich często posiada własne, dodatkowe zależności. O ile taka sytuacja nie występuje, ręczne zestawienie takiego drzewka jest w zupełnie wystarczające i na codziennym porządku. Jednakże w przypadku „zagnieżdżonych” zależności takie ręczne zestawienie byłoby nie lada wyzwaniem, a także niełatwym w utrzymaniu. Jeśli używamy menadżera paczek zwanego *npm*, z pomocą przychodzi nam komenda *npm ls*, która pozwala nam wyświetlić wszystkie zewnętrzne zależności oraz podzależności naszej aplikacji w postaci drzewka, co przedstawione jest na Rysunku 5.5. Drzewa zależności są bardzo ważne w inżynierii oprogramowania, ponieważ pozwalają na ocenę kodu nadzędnej aplikacji, co może skutkować drastycznym zmniejszeniem rozmiaru pliku oraz ilości zużywanej pamięci.

```
[→ app git:(main) npm ls
app@0.1.0 /Users/rafonix/video-chat/app
├── @typescript-eslint/eslint-plugin@4.29.3
├── @typescript-eslint/parser@4.29.3
├── @vue/cli-plugin-babel@4.5.13
├── @vue/cli-plugin-eslint@4.5.13
├── @vue/cli-plugin-typescript@4.5.13
├── @vue/cli-service@4.5.13
├── @vue/eslint-config-typescript@7.0.0
├── axios@0.21.1
├── babel-eslint@10.1.0
├── core-js@3.16.3
├── eslint-plugin-vue@6.2.2
├── eslint@6.8.0
├── firebase@9.0.2
├── sass-loader@10.2.0
├── sass@1.32.13
├── typescript@4.1.6
├── vee-validate@3.4.13
├── vue-class-component@7.2.6
├── vue-cli-plugin-vuetify@2.4.2
├── vue-property-decorator@9.1.2
├── vue-router@3.5.2
├── vue-socket.io@3.0.10
├── vue-template-compiler@2.6.14
├── vue@2.6.14
├── vuetify-loader@1.7.3
├── vuetify@2.5.8
└── vuex-module-decorators@1.0.1
   └── vuex@3.6.2
```

Rysunek 5.5 Przykład użycia *npm ls* w mojej aplikacji

Duże aplikacje zazwyczaj posiadają zależności o długości nawet dziesięciu tysięcy elementów. Ręczna ocena takich zależności jest praktycznie niemożliwa do wykonania. Dlatego też w takich przypadkach niezbędne okażą się zautomatyzowane skrypt wykonujące taką czynność za nas oraz inne techniki zapewniające integralność zależności, na których polegamy. Tak, przygotowane przez skrypty, drzewka zależności możemy następnie zestawić z bazą powszechnie znanych i zarejestrowanych podatności, np. *CVE*, aby zidentyfikować czy nie otwierają one (zależności) luk bezpieczeństwa w naszej aplikacji. Do zbudowania drzewa zależności aplikacji można też użyć różnego rodzaju skanera zewnętrznych zależności tj. *Snyk*.

Kolejnym sposobem radzenia sobie z potencjalnymi niebezpieczeństwami niesionymi przez zewnętrzne zależności jest oddzielenie głównego serwera aplikacji i umieszczenie zależności na osobnym serwerze. W tak przygotowanej konfiguracji, nasza aplikacja będzie się jedynie komunikować z serwerem zależności za pomocą protokołu *HTTP* oraz wymieniać informacje w postaci *JSON*, który znacznie ogranicza wykonywanie skryptów (niechcianych) na głównym serwerze aplikacji. Jednakże rozwiązanie takie nie pozostaje bez wad. Jedną z nich jest to, że jakiekolwiek wrażliwe dane wysłane na serwer z zależnościami nadal mogą zostać zmodyfikowane w niechciany przez nas sposób. Ponadto, rozwiązanie to w pewnym stopniu zmniejszy wydajność naszej aplikacji poprzez ciągłe zapytania do oddzielnego serwera (zależności).

Na koniec warto również zwrócić uwagę na menadżery paczek. Można rzec, iż praktycznie większość aplikacji zbudowanych w technologii *JavaScript* używa specjalnie stworzonego dla tego języka menadżera paczek – *npm*. Menadżery takie z reguły nie są najbezpieczniejsze i już nie raz w historii doprowadzały nawet wielkie firmy do zhackowania. Jednym sposobem na zabezpieczenie takich menadżerów jest indywidualne przeprowadzanie audytów pobieranych przez nich paczek a następnie blokowanie aplikacji do używania tylko tych wersji paczek, które zostały uznane po skończonym audycie za bezpieczne. Niestety, nawet tak ręcznie zablokowane paczki, mogą zostać automatycznie zaktualizowane do najnowszej wersji bez wiedzy osób ich używających. Nowe wersje, często jeszcze dobrze nieprzetestowane, mogą otwierać nowe luki bezpieczeństwa w aplikacjach, w których są stosowane. Rozwiązaniem tego problemu jest wykonanie komendy *npm shrink wrap* w katalogu naszego projektu. Komenda ta pobierze wszystkie zależności oraz podzależności tych zależności w obecnej wersji, ale i także bezwzględnie zablokuje te wersje tak, że nie zostaną one z pewnością automatycznie zaktualizowane. [1]

## **6. Testy zastosowanych zabezpieczeń**

Po zebraniu informacji na temat budowy architektury nowoczesnych aplikacji internetowych, potencjalnych jej zagrożeń oraz ataków jakie można na nią wykonać, a także po przedstawieniu sposobów obrony przed wspomnianymi atakami przyszedł czas na rzeczywiste testy na realnej aplikacji sprawdzające, czy aby na pewno informacje, zawarte w części teoretycznej pracy, mają swoje zastosowanie w rzeczywistości. W niniejszym rozdziale zostanie przeprowadzony szereg testów sprawdzających podatności oraz przykładowe sposoby im zapobiegania przy użyciu wystawionej aplikacji typu komunikator internetowy. Następnie zebrane wyniki testów zostaną poddane analizie i na ich podstawie wyciągnięte będą wnioski, skorygowane z założeniami teoretycznymi pracy.

### **6.1. Wystawienie aplikacji do sieci lokalnej**

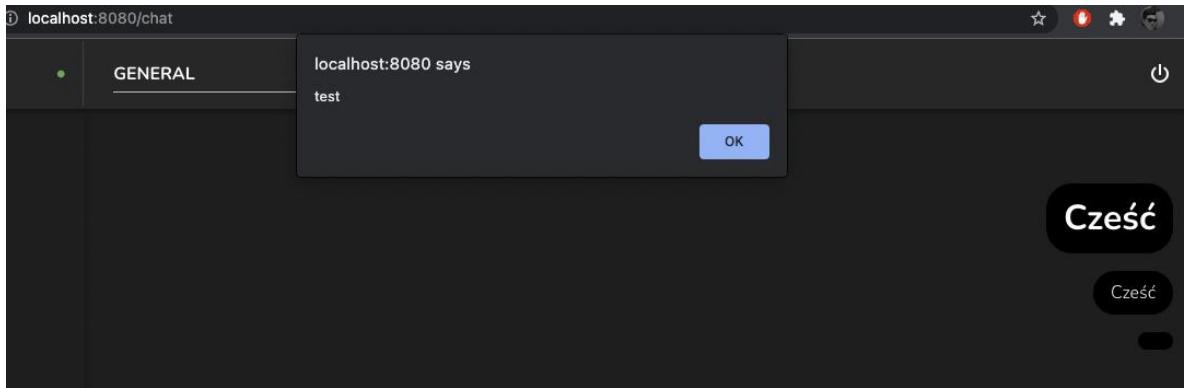
W celu przetestowania aplikacji pod kątem bezpieczeństwa tj. przeprowadzenia szeregu ataków na nią, postanowiłem wystawić jej serwisy do sieci lokalnej, aby móc ją tam przetestować. Strona klienta aplikacji została wystawiona na *localhost* na porcie 8080 natomiast serwer *API* działa tak samo na *localhost*, ale za to na porcie 3000. Dodatkowymi serwisami niezbędnymi do prawidłowego działania aplikacji jest serwis bazy danych – *MySQL* oraz bazy danych do szybkiego odczytu – *Redis*. Serwisy te działają odpowiednio na portach 3306 oraz 6379. Ponadto, na potrzeby rejestracji przebiegu testów oraz wykonywanych żądań, zaimplementowałem logowanie akcji wykonywanych na serwerze. Wszelkie akcje zostają zapisywane do pliku *access.log*, którego zawartość czyszczona jest co każdą następną dobę, a poprzednie logi zostają zarchiwizowane. [18]

### **6.2. Testowanie aplikacji**

W podrozdziale tym zostanie wykonane pięć rodzajów ataków na komunikator internetowy, zawartych w części teoretycznej pracy. Ataki, jeśli możliwe, zostaną przeprowadzone na różne sposoby. W przypadku odkrycia podatności, zostanie ona odpowiednio zabezpieczona, a następnie zostanie dokonany ten sam atak sprawdzający, czy aby na pewno podatność została unieważniona.

#### **6.2.1. Przeprowadzenie ataków**

Jako pierwszy pod uwagę weźmiemy najbardziej powszechny atak *XSS*, czyli *Stored XSS*. W związku z tym, że nasza aplikacja to komunikator, użytkownicy są w stanie między sobą wymieniać własnoręcznie napisane wiadomości, które następnie są zapisywane do bazy oraz za każdym razem wyświetlane w przeglądarce. Sytuacja ta tworzy wręcz idealny scenariusz do przeprowadzenia ataku typu *Stored XSS*, ponieważ atak ten polega właśnie na przechowywaniu złośliwego skryptu na bazie danych i wykonywaniu go za każdym razem, gdy użytkownik pobierze z bazy dane zawierające ten skrypt. Warto zatem sprawdzić, czy wysyłane wiadomości są na pewno sprawdzane pod kątem zawartości.



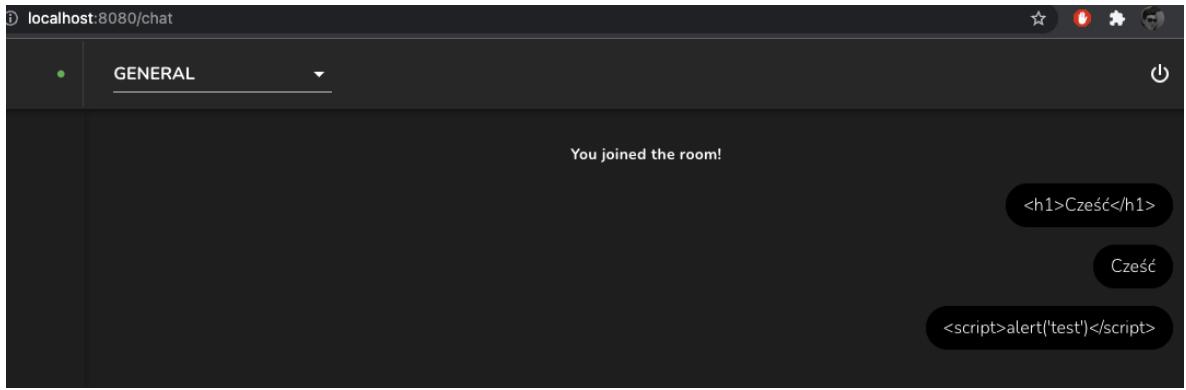
Rysunek 6.1 Pomyślne wykonanie ataku typu Stored XSS

Niestety, formularz do wysyłania wiadomości nie jest odpowiednio zabezpieczony, gdyż pomyślnie wykonuje wprowadzone wiadomości jako kod *HTML*, bądź *JavaScript*. Jak widzimy, pierwsza wysłana wiadomość jest napisana większą czcionką niż druga oraz jest pogrubiona, ponieważ oprócz samej treści, została również ona opakowana w znaczniki *<h1></h1>*, które powodują stworzenie nagłówka w języku *HTML*. Trzecia wiadomość natomiast jest opakowana w znaczniki *<script></script>*, które powodują wykonanie skryptu znajdującego się między nimi – stąd brak treści wiadomości, gdyż między znacznikami umieszczony został skrypt *alert('test')* powodujący pojawienie się okienka na wyżej przedstawionym Rysunku 6.1. Co gorsza skrypt ten jest zapisany w strukturze wiadomości, która jest potem wysyłana na bazę danych, a więc będzie on zawsze wykonywany po wejściu użytkownika w dane wiadomości.

```
▼<p class="py-2 px-4 rounded-xl chat-area__my-message">
  ▼<span>
    <script>alert('test')</script> = $0
  </span>
```

Rysunek 6.2 Umieszczony kod w DOM aplikacji po przeprowadzeniu ataku XSS

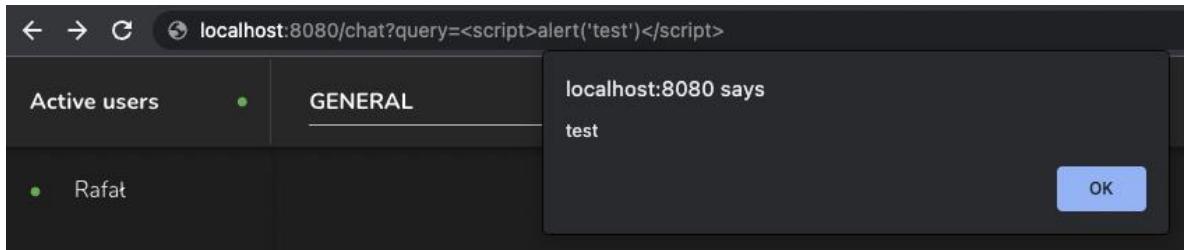
Skrypt umieszczony w wiadomości może być znacznie bardziej szkodliwy niż ten podany tutaj w celach demonstracyjnych. Powodem tego problemu jest użycie w aplikacji metody *innerHTML* do budowania struktury wiadomości. Metoda ta „dokleja” do struktury strony cały podany jej kod, stąd efekt przedstawiony na Rysunku 6.2. Należy zmienić tą metodę na *innerText* odpowiadającą za przedstawianie wpisywanej wiadomości jako ciągu znaków, co zneutralizuje wyżej napotkaną podatność.



Rysunek 6.3 Efekt zabezpieczenia aplikacji przed Stored XSS

Po zastosowaniu odpowiedniej metody, podatność zostaje zneutralizowana, a wszelkie próby umieszczenia jakiegokolwiek wykonywalnego kodu kończą się na zapisaniu go w bezpiecznej postaci ciągu znaków – ciąg znaków nigdy się nie wykona, a jedynie będzie jawnym tekstem, co ukazane jest na Rysunku 6.3.

Kolejny atak XSS, typu *Reflected*, polega na odpowiednim spreparowaniu linku tak, aby użytkownik po wejściu w dany link nieumyślnie wykonał zawarty w nim skrypt. Dla celów demonstracyjnych skrypt zawarty w url będzie jak najprostszy tak, aby przedstawić jedynie jego działanie.

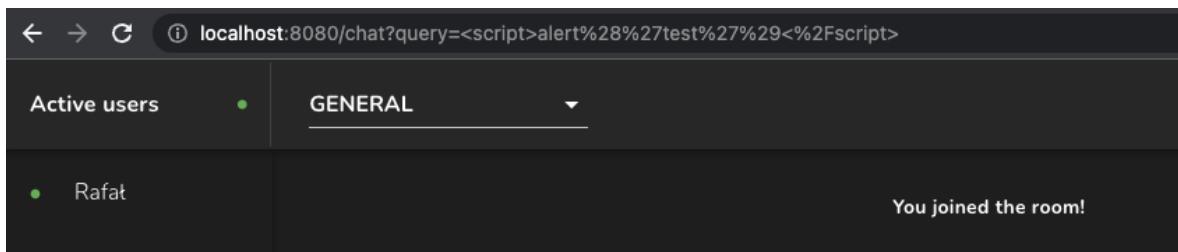


Rysunek 6.4 Pomyślne wykonanie ataku typu Reflected XSS

Jak widzimy na Rysunku 6.4, logika odpowiadająca za przekierowanie wykonała skrypt zawarty w parametrze zapytania naszego linku. Oczywiście można byłoby tam dodać kod bardziej szkodzący aplikacji, np. wyciągnięcie danych z bazy poprzez zapytanie do niej. Dlatego tak ważne jest, aby zabezpieczyć aplikację przez atakami typu *Reflected XSS*. Jednym sposobem, który osobiście zaimplementowałem, jest oczyszczanie url'a z niedozwolonych znaków. Dokonałem tego za pomocą wbudowanej w *JavaScript* funkcji *encodeURI*:

```
const sanitizedUrl = encodeURI(url);
```

Metoda ta „oczyszcza” link z niedozwolonych znaków poprzez kodowanie na ich bezpieczne odpowiedniki. Po zastosowaniu tej strategii bezpieczeństwa, wejście we wcześniej spreparowany oraz niebezpieczny link nie grozi wykonaniem jakiegokolwiek skryptu, ponieważ znaki odpowiadające za umieszczenie tego skryptu zostaną zakodowane, jak z resztą widać na Rysunku 6.5 przedstawionym poniżej:



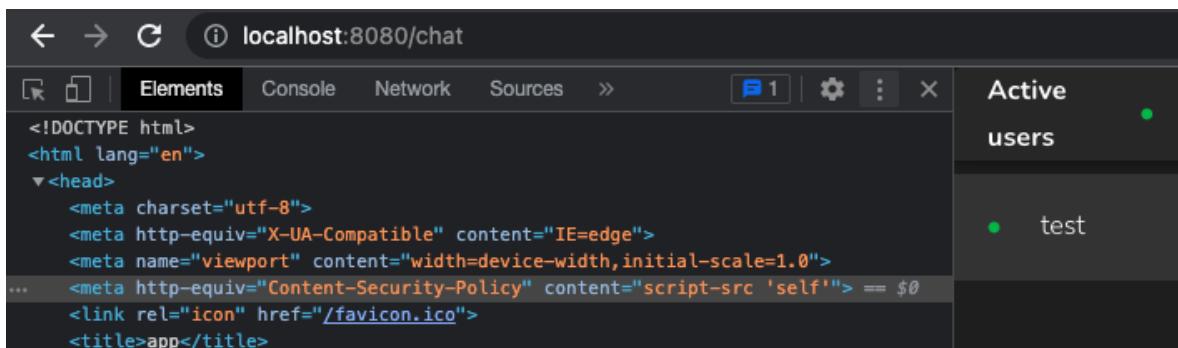
Rysunek 6.5 Efekt zabezpieczenia aplikacji przed Reflected XSS

Ostatnim omawianym typem ataku XSS jest *DOM-based XSS*. Najczęstszym miejscem wyprowadzania tego ataku jest modyfikacja parametrów *URL* aplikacji, a następnie, za pomocą kodu, aplikacja wprowadza logikę ze zmodyfikowanego linku do struktury strony. Napisany na potrzeby pracy komunikator został zbudowany w oparciu o nowe technologie oraz frameworki, w związku z czym aplikacja, w celu dynamicznego pobierania danych, nie wykorzystuje parametrów zapytania, a wysyła żądania do serwera *API*. W związku z tym nie posiada jednoznacznej podatności na tego typu atak i jest z góry bezpieczna w tym rejonie. Jednakże każda aplikacja się rozwija, więc istnieje możliwość otwarcia takiej luki bezpieczeństwa w przyszłości. Aby do tego nie dopuścić, należałoby uważać by nie używać funkcji bezpośrednio wykonujących ciąg znaków jako kod, takich jak *eval()* czy wcześniej wspomniany *innerHTML*.

Dobrą praktyką na zabezpieczenie się przed atakami XSS jest też konfiguracja *CSP* (*Content Security Policy*). Przy poprawnej konfiguracji *CSP* nawet jeśli atakujący, naszą aplikację, znajdzie podatność na atak XSS, prawdopodobnym jest, iż nie będzie mógł jej wykorzystać, gdyż *CSP* zablokuje wykonywanie jakiegokolwiek kodu z zewnątrz. W aplikacji postanowiłem, w celach demonstracyjnych, zaimplementować jak najbardziej podstawową konfigurację *CSP*, blokującą właśnie skrypt z jakiejkolwiek innej domen niż ta, która „hostuje” moją aplikację.

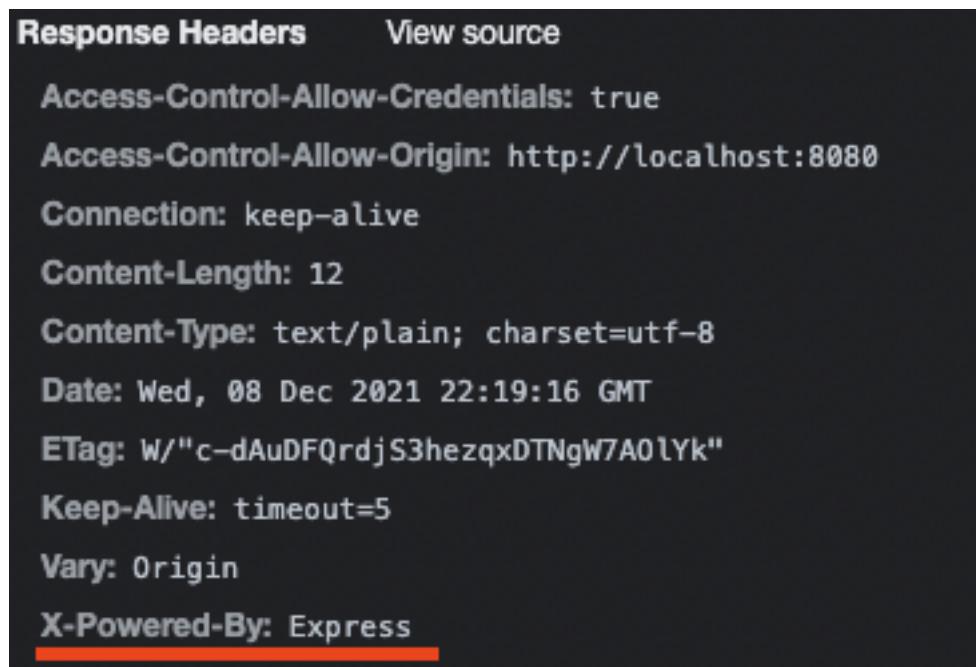
```
<meta http-equiv="Content-Security-Policy" content="script-src 'self'"/>
```

Aby sprawdzić, czy *CSP* zostało zaimplementowane poprawnie w aplikacji, należy, w narzędziach developerskich przeglądarki, wyszukać odpowiedniego meta tagu – *Content-Security-Policy*. Gdy ten *meta tag* zostanie pomyślnie odnaleziony, oznacza to, że *CSP* zostało zaimplementowane pomyślnie oraz działa. Rysunek 6.6 przedstawia poprawnie odnaleziony szukany *tag*.



Rysunek 6.6 Pomyślne odnalezienie CSP w DOM przeglądarki

Warto również pozbyć się niepożądanych nagłówków wysyłanych domyślnie przez serwer zawierających informacje na przykład o tym, jaki framework jest wykorzystywany w aplikacji.



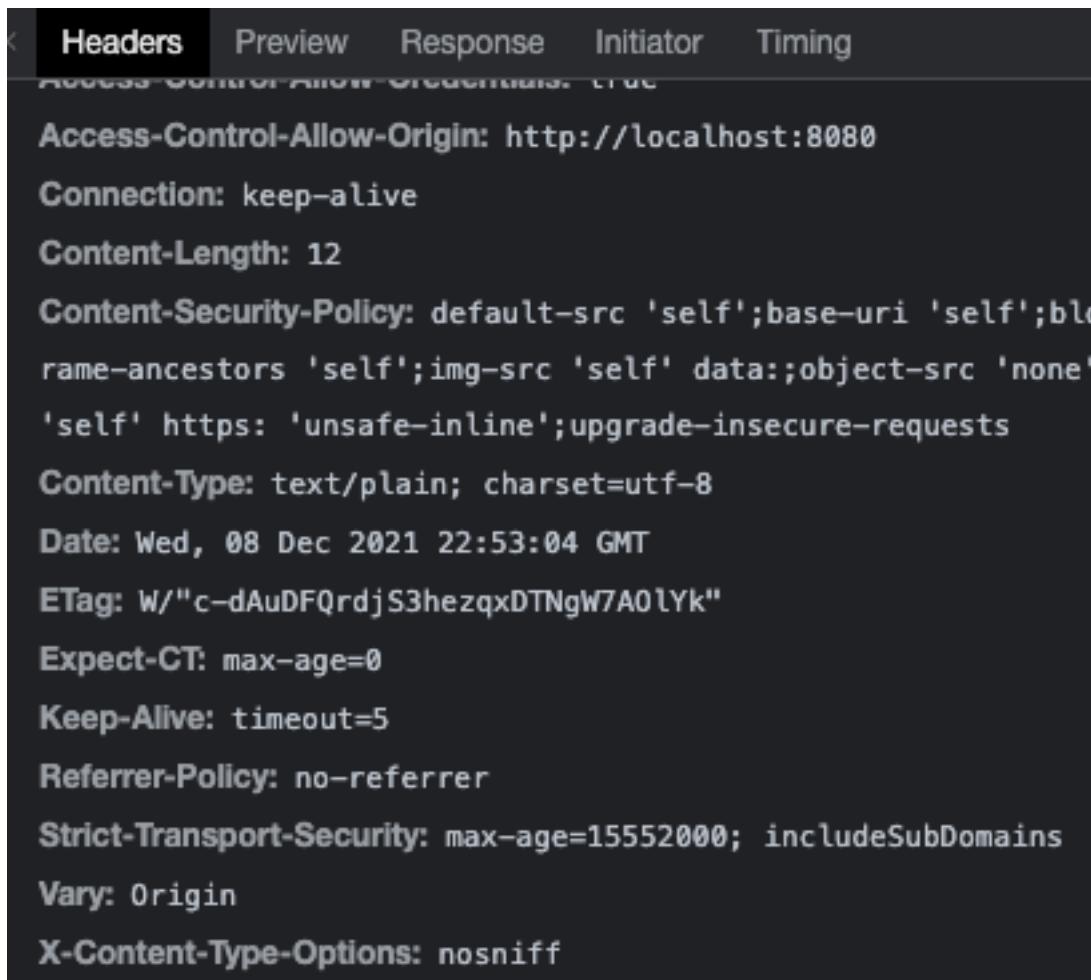
The screenshot shows the 'Response Headers' section of a browser's developer tools. It lists several headers with their values:

- Access-Control-Allow-Credentials:** true
- Access-Control-Allow-Origin:** http://localhost:8080
- Connection:** keep-alive
- Content-Length:** 12
- Content-Type:** text/plain; charset=utf-8
- Date:** Wed, 08 Dec 2021 22:19:16 GMT
- ETag:** W/"c-dAuDFQrdjS3hezqxDTNgW7A01Yk"
- Keep-Alive:** timeout=5
- Vary:** Origin
- X-Powered-By:** Express

Rysunek 6.7 Nagłówki odpowiedzi serwera

Serwer mojej aplikacji jest postawiony na *Node.js* na frameworku *Express*, co jest niestety widoczne w nagłówkach, jak przedstawia Rysunek 6.7. Potencjalny atakujący bez większego wysiłku może otrzymać oraz wykorzystać tą informację do przeprowadzenia ataku na aplikację, np. sprawdzając już odkryte podatności frameworku *Express* w bazie CVE. Aby pozbyć się niepożądanych nagłówków, skorzystałem z biblioteki *helmet*, która takowe nagłówki odfiltrowuje. [19][20]

```
const helmet = require("helmet");
app.use(helmet());
```



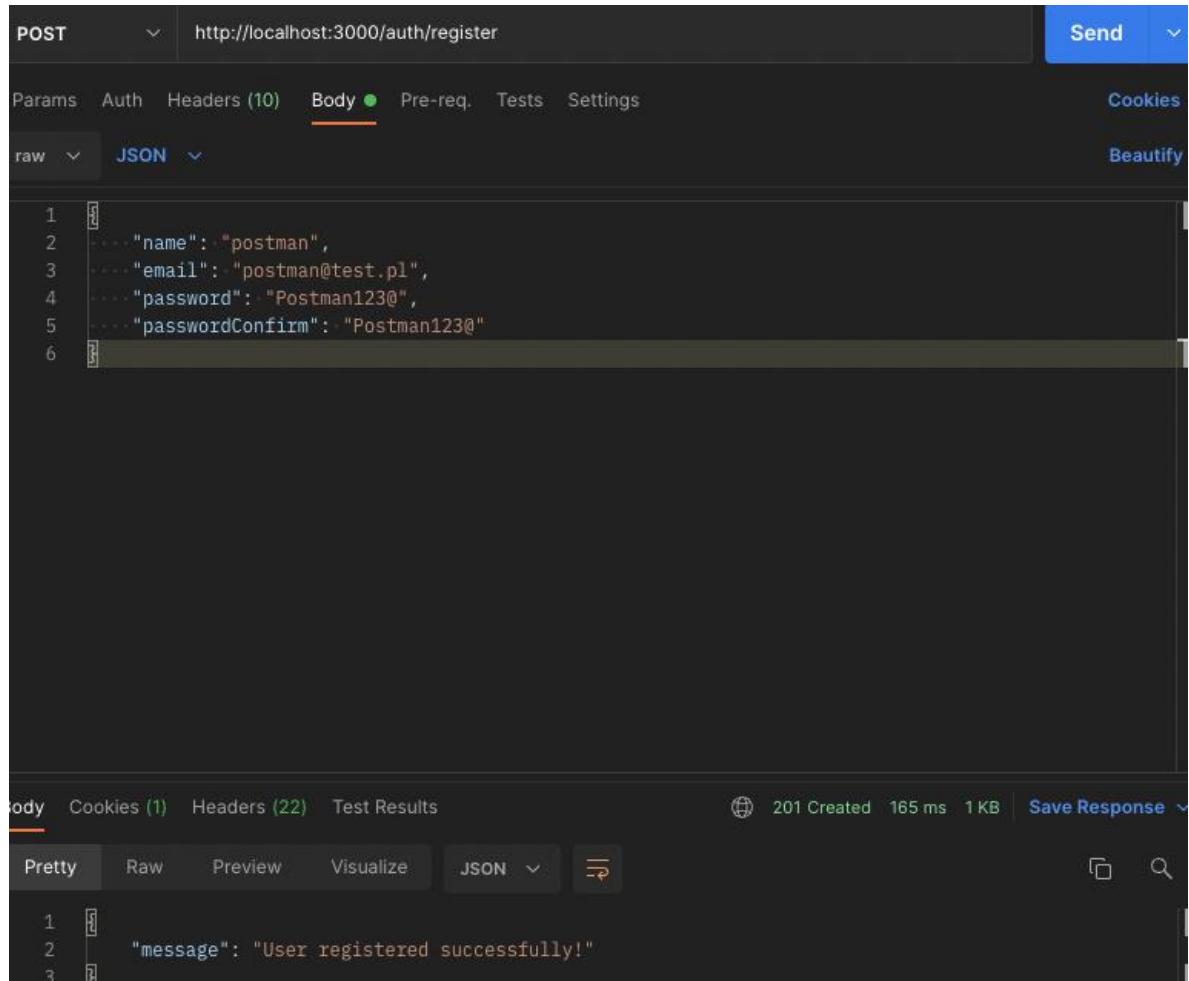
The screenshot shows the 'Headers' tab of the Postman interface. The response section displays the following headers:

- Access-Control-Allow-Origin:** http://localhost:8080
- Connection:** keep-alive
- Content-Length:** 12
- Content-Security-Policy:** default-src 'self';base-uri 'self';block-all-mixed-content;child-frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
- Content-Type:** text/plain; charset=utf-8
- Date:** Wed, 08 Dec 2021 22:53:04 GMT
- ETag:** W/"c-dAuDFQrdjS3hezqxDTNgW7A0lYk"
- Expect-CT:** max-age=0
- Keep-Alive:** timeout=5
- Referrer-Policy:** no-referrer
- Strict-Transport-Security:** max-age=15552000; includeSubDomains
- Vary:** Origin
- X-Content-Type-Options:** nosniff

Rysunek 6.8 Zabezpieczone nagłówki odpowiedzi serwera

Jak można zauważyć na Rysunku 6.8 powyżej, nagłówek z informacją o używanym frameworku serwera nie jest już wysyłany, a dodatkowo, za pomocą biblioteki *helmet*, wysyłane są dodatkowe, zabezpieczające nagłówki, których omówienie wykracza już poza zakres tej pracy.

Aby zabezpieczyć swoją aplikację przed atakami *CSRF*, postanowiłem zaimplementować jedną z najbardziej skutecznych metod obrony przed nimi, czyli *CSRF cookies*. Do tej pory wykonywanie skryptów z zewnątrz aplikacji, na przykład za pośrednictwem skradzionych danych osoby autoryzowanej, było dla osób trzecich możliwe. W celu udowodnienia postawionej tezy, przeprowadzę rejestrację z zewnątrz aplikacji za pośrednictwem narzędzia *Postman*, aby dokonać próby rejestracji użytkownika.



Rysunek 6.9 Rejestracja nowego użytkownika za pomocą narzędzia Postman

Rysunek 6.9 dowodzi temu, że nieautoryzowane osoby z zewnątrz mogą, po odpowiednim przygotowaniu zapytania, dokonywać dowolne akcje w testowanej aplikacji. Nie byłoby też problemem podrzucić zautoryzowanej ofierze odpowiednio spreparowany link z żądaniem, aby ta dokonała tego żądania w naszym imieniu. Aby temu zaradzić, zaimplementowałem odpowiednią funkcję, używając biblioteki *csurf*, która, do wykonania jakichkolwiek żądań *PUT*, *POST*, *DELETE* itd., sprawdza czy w żądaniu zawarty jest token *CSRF* oraz czy jest on poprawny czy ważny.

```
const csrf = require("csurf");
const csrfProtection = csrf({ cookie: true });

app.use(csrfProtection);
```

The screenshot shows a POST request to `http://localhost:3000/auth/register`. The request body is JSON, containing:

```
1  {
2    ... "name": "postman",
3  ... "email": "postman@test.pl",
4  ... "password": "Postman123@",
5  ... "passwordConfirm": "Postman123@"
6 }
```

The response status is 403 Forbidden, with a message: `ForbiddenError: invalid csrf token`.

Rysunek 6.10 Próba rejestracji użytkownika za pomocą narzędzia Postman

Na Rysunku 6.10 widzimy, że dokonanie rejestracji użytkownika z zaimplementowanym zabezpieczeniem punktów końcowych aplikacji za pomocą tokenów *CSRF* jest teraz niemożliwe. Niepowodzenie rejestracji możemy zaobserwować również w logach serwera na Rysunku 6.11. Widać tam, między innymi, że żądanie posiada status 403 (odmowa dostępu) oraz zostało wykonane za pośrednictwem narzędzia *Postman*.

```
::1 - - [08/Dec/2021:23:40:38 +0000] "POST /auth/register HTTP/1.1" 403 2168 "-" "PostmanRuntime/7.28.3"
```

Rysunek 6.11 Logi z serwera wskazujące na nieudane żądanie rejestracji

Aby dokonać pomyślnej rejestracji, należy przy każdym żądaniu wysyłać token *CSRF*, który jest unikalny dla każdego użytkownika tworzącego sesję z aplikacją – oznacza to, że nikt z zewnątrz nie będzie w stanie za naszym pośrednictwem dokonać jakichkolwiek akcji w aplikacji. Jedynym rozwiązaniem dla osoby atakującej jest wykradnięcie naszego tokenu *CSRF*, co i tak znacznie ogranicza możliwości ataku, ponieważ aby przeprowadzić atak na dużą skalę, należałoby wykraść tysiące ważnych tokenów. Rysunek 6.12 przedstawia próbę żądania za pośrednictwem aplikacji *Postman* w celu otrzymania ważnego tokenu *CSRF*, a następnie – przy jego użyciu – dokonać ponownej próby rejestracji użytkownika.

The screenshot shows the Postman interface with a GET request to `http://localhost:3000/auth/csrf-token`. The 'Params' tab is selected, showing a single parameter `_csrf` with the value `3z5Aefj1-37MPyuCexkLC1lFwWrIk-YH65pY`. The response status is 200 OK, and the response body is a JSON object with a single key `"csrfToken": "3z5Aefj1-37MPyuCexkLC1lFwWrIk-YH65pY"`.

Rysunek 6.12 Otrzymanie ważnego CSRF token za pomocą narzędzia Postman

The screenshot shows the Postman interface with a POST request to `http://localhost:3000/auth/register?_csrf=3z5Aefj1-37MPyuCexkLC1lFwWrIk-YH65pY`. The 'Params' tab is selected, showing a parameter `_csrf` with the value `3z5Aefj1-37MPyuCexkLC1lFwWrIk-YH65pY`. The response status is 201 Created, and the response body is a JSON object with a key `"message": "User registered successfully!"`.

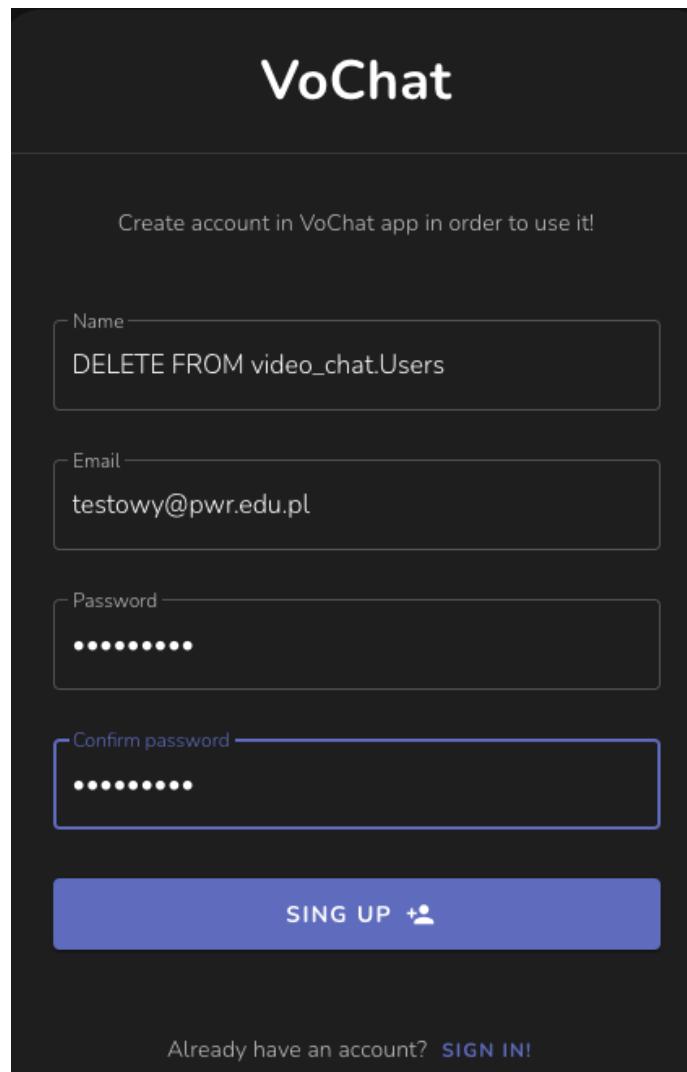
Rysunek 6.13 Pomyślna próba rejestracji użytkownika przy użyciu ważnego tokenu CSRF

Dopiero przy wysłaniu w parametrach żądania prawidłowego oraz ważnego tokenu *CSRF* byłem w stanie zarejestrować poprawnie użytkownika, co widać na Rysunku 6.13. Dowodzi to poprawności działania zaimplementowanego mechanizmu do obrony przed atakami *CSRF*. W taki oto sposób wiemy, że skuteczność jakiejkolwiek formy ataku *CSRF*, na naszą aplikację, jest w dużym stopniu ograniczona. Pomyślną rejestrację użytkownika wraz z otrzymaniem ważnego tokenu możemy też zaobserwować w logach aplikacji, co przedstawiono na Rysunku 6.14.

```
::1 -- [08/Dec/2021:23:56:46 +0000] "GET /auth/csrf-token HTTP/1.1" 200 52 "-" "PostmanRuntime/7.28.3"  
::1 -- [08/Dec/2021:23:59:03 +0000] "POST /auth/register?_csrf=3z5Aefj1-37MPyuCexkLC11FwWrIk-YH65pY HTTP/1.1" 201 43 "-" "PostmanRuntime/7.28.3"
```

Rysunek 6.14 Wpisy w logach wskazujące na otrzymanie ważnego tokenu oraz pomyślną rejestrację

Ataki *SQL Injection* były bardzo popularne, a zarazem niebezpieczne w przeciągu ostatniej dekady. Jednakże są one łatwe do zapobiegnięcia, a dodatkowo rozwój technologii pozwolił w łatwy sposób im zapobiec. Aplikacja, którą stworzyłem, oparta jest na nowym i zaprojektowanym w sposób bezpieczny framework'u serwerowym – *Express.js*, a budowanie zapytań na bazę zaimplementowałem za pomocą mapowania obiektowo-relacyjnego, czyli sposobu odwzorowania obiektowej architektury na bazę danych *SQL*. Sposób ten używa spараметryzowanych zapytań, tak więc aplikacja dzięki temu jest całkowicie zabezpieczona przed atakiem *SQL Injection*. Aby potwierdzić tezę, wykonam test rejestracji użytkownika z próbą wrzucenia zapytania do bazy powodującego teoretycznie usunięcie tabeli użytkowników. [21]

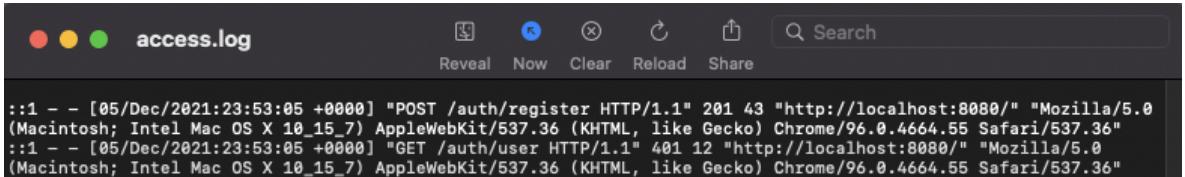


Rysunek 6.15 Dane w formularzu potrzebne do rejestracji użytkownika

```
▼ Request Payload      view source
  ▼ {name: "DELETE FROM video_chat.Users", email: "testowy@pwr.edu.pl"
    email: "testowy@pwr.edu.pl"
    name: "DELETE FROM video_chat.Users"}
```

Rysunek 6.16 Dane wysłane na serwer

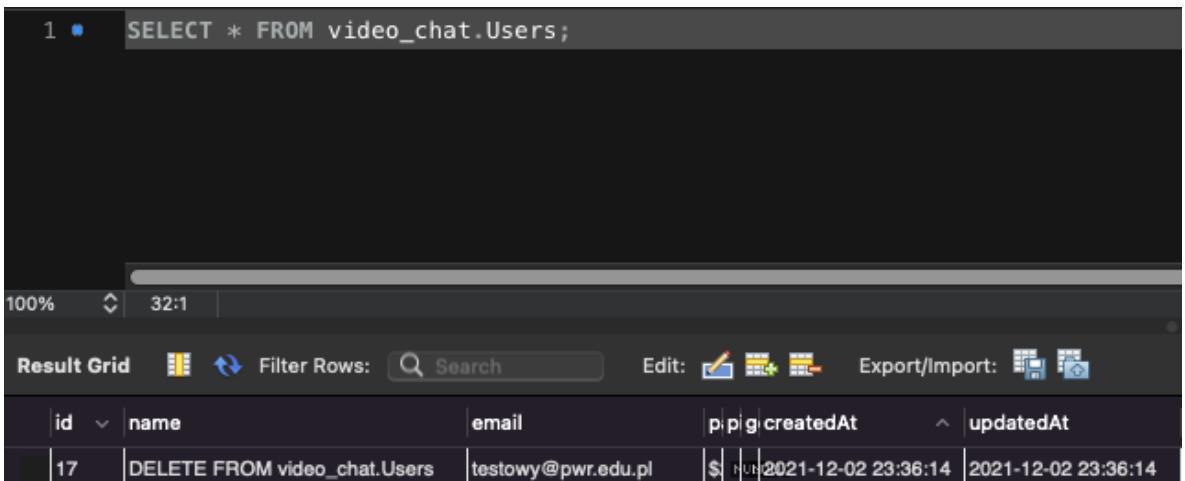
Jak widzimy na załączonych Rysunkach 6.15 oraz 6.16, rejestracja użytkownika o nazwie zapytania do bazy powodującego usunięcie całego rekordu z użytkownikami aplikacji przebiegła pomyślnie. Do serwera zostały przesłane dane do rejestracji w postaci obiektu z kluczami zawierającymi ciągi znaków z niezbędnymi informacjami. Możemy także zauważyc nowy wpis w logach informujący nas o żądaniu do punktu końcowego odpowiedzialnego za rejestrację użytkownika. Wpis w logu, przedstawiony na Rysunku 6.17, dostarcza nam wielu przydatnych informacji tj. adres IP, czas wykonania żądania czy nawet przeglądarki użytej w celu wykonania wspomnianego żądania.



```
access.log
Reveal Now Clear Reload Share Search
::1 - - [05/Dec/2021:23:53:05 +0000] "POST /auth/register HTTP/1.1" 201 43 "http://localhost:8080/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.55 Safari/537.36"
::1 - - [05/Dec/2021:23:53:05 +0000] "GET /auth/user HTTP/1.1" 401 12 "http://localhost:8080/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.55 Safari/537.36"
```

Rysunek 6.17 Wpis w logach informujący nas o wykonaniu żądania rejestracji użytkownika

Następnym krokiem jest sprawdzenie, czy aby przypadkiem „nazwa” nowo utworzonego użytkownika nie usunęła tabeli w bazie danych.



id	name	email	password	createdAt	updatedAt
17	DELETE FROM video_chat.Users	testowy@pwr.edu.pl	\$	2021-12-02 23:36:14	2021-12-02 23:36:14

Rysunek 6.18 Stan bazy danych po przeprowadzonym teście podatności na SQL Injection

Dowodem potwierdzenia tezy jest Rysunek 6.18 - rekord użytkownika, tworzonego w celach demonstracyjnych, został zapisany pomyślnie bez jakichkolwiek negatywnych skutków ubocznych. Jak możemy zauważyć, nazwa tego użytkownika (a'la złośliwe zapytanie do bazy) została zapisana w postaci tekstu, co potwierdza bezpieczeństwo używania mapowania obiektowo-relacyjnego oraz zawartych w nim mechanizmów bezpieczeństwa dot. *SQL Injection*, tj. przykładowo parametryzowanie zapytań, których prawidłowe działania możemy sprawdzić już we wpisach w rejestrze serwera wyświetlającym wykonywane działania na bazie danych.

```
Executing (default): SELECT `id`, `name`, `email`, `password`, `picture`, `googleId`, `createdAt`, `updatedAt` FROM `Users` AS `User` WHERE `User`.`email` = 'testowy@pwr.edu.pl' LIMIT 1;
Executing (default): INSERT INTO `Users` (`id`, `name`, `email`, `password`, `createdAt`, `updatedAt`) VALUES (DEFAULT, ?, ?, ?, ?, ?);
```

Rysunek 6.19 Rejestr serwera zawierający informacje o wykonywanych działaniach na bazie danych

Jak widzimy w przedostatniej linijce Rysunku 6.19, wszystkie wartości wrzucane bezpośrednio na bazę zostały sparametryzowane, co skutecznie zapobiega *SQL Injection*. Jest to również dowód na bezpieczeństwo stosowania mapowania obiektowo-relacyjnego.

W przypadku ataków *DoS*, należy mieć na uwadze trzy jego rodzaje: *Regex DoS*, logiczny *DoS* oraz *DDoS*. W przypadku pierwszego, jego zapobieganie jest stosunkowo łatwe, ponieważ wystarczy wyszukać miejsc w aplikacji, które są podatne na złośliwe wyrażenia regularne, czyli takie które zawierają między innymi następujące znaki: „+” czy „\*”. Następnie należy takie miejsca wyeliminować bądź zaimplementować logikę

zabezpieczającą wyrażenia regularne przed atakiem *DoS*. Aby znaleźć miejsca w aplikacji podatne na ataki *Regex DoS*, przeskanowałem ją przy użyciu narzędzia *npm audit*.

```
→ app git:(main) ✘ npm audit
    └─── npm audit security report ───

# Run npm install --save-dev eslint@8.4.0 to resolve 1 vulnerability
SEMVER WARNING: Recommended action is a potentially breaking change



|               |                                                                                                                   |
|---------------|-------------------------------------------------------------------------------------------------------------------|
| Moderate      | Inefficient Regular Expression Complexity in chalk/ansi-regex                                                     |
| Package       | ansi-regex                                                                                                        |
| Dependency of | eslint [dev]                                                                                                      |
| Path          | eslint > strip-ansi > ansi-regex                                                                                  |
| More info     | <a href="https://github.com/advisories/GHSA-93q8-gq69-wqmw">https://github.com/advisories/GHSA-93q8-gq69-wqmw</a> |



# Run npm update ansi-regex --depth 10 to resolve 13 vulnerabilities


|               |                                                                                                                   |
|---------------|-------------------------------------------------------------------------------------------------------------------|
| Moderate      | Inefficient Regular Expression Complexity in chalk/ansi-regex                                                     |
| Package       | ansi-regex                                                                                                        |
| Dependency of | @vue/cli-service [dev]                                                                                            |
| Path          | @vue/cli-service > cli-highlight > yargs > string-width > strip-ansi > ansi-regex                                 |
| More info     | <a href="https://github.com/advisories/GHSA-93q8-gq69-wqmw">https://github.com/advisories/GHSA-93q8-gq69-wqmw</a> |


```

Rysunek 6.20 Wynik skanu aplikacji w celu wyszukania miejsc podatnych na Regex DoS

Raport ze skanu, ukazany na Rysunku 6.20, bezpośrednio wskazuje nam, że w naszej aplikacji istnieje 13 podatności na ataki *Regex DoS*. Te luki bezpieczeństwa pochodzą głównie z zależności zewnętrznych, które będę testował oraz zabezpieczać w kolejnym kroku. Na ten moment jednak należy pozbyć się znalezionych podatności. Możemy to w łatwy sposób zrobić za pomocą komendy *npm update ansi-regex --depth 10*.

```
→ app git:(main) ✘ npm update ansi-regex --depth 10
npm notice created a lockfile as package-lock.json. You should commit this file.
+ ansi-regex@5.0.1
updated 1 package and audited 3 packages in 0.331s
found 0 vulnerabilities

npm notice created a lockfile as package-lock.json. You should commit this file.
+ ansi-regex@5.0.1
added 1 package from 1 contributor, updated 1 package and audited 7 packages in 0.322s
found 0 vulnerabilities

npm notice created a lockfile as package-lock.json. You should commit this file.
+ ansi-regex@5.0.1
added 1 package from 1 contributor, updated 1 package and audited 13 packages in 0.291s

2 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities

npm notice created a lockfile as package-lock.json. You should commit this file.
+ ansi-regex@5.0.1
added 1 package from 1 contributor, updated 1 package and audited 10 packages in 0.283s

1 package is looking for funding
  run 'npm fund' for details

found 0 vulnerabilities

npm notice created a lockfile as package-lock.json. You should commit this file.
+ ansi-regex@5.0.1
added 1 package from 1 contributor, updated 1 package and audited 13 packages in 0.314s

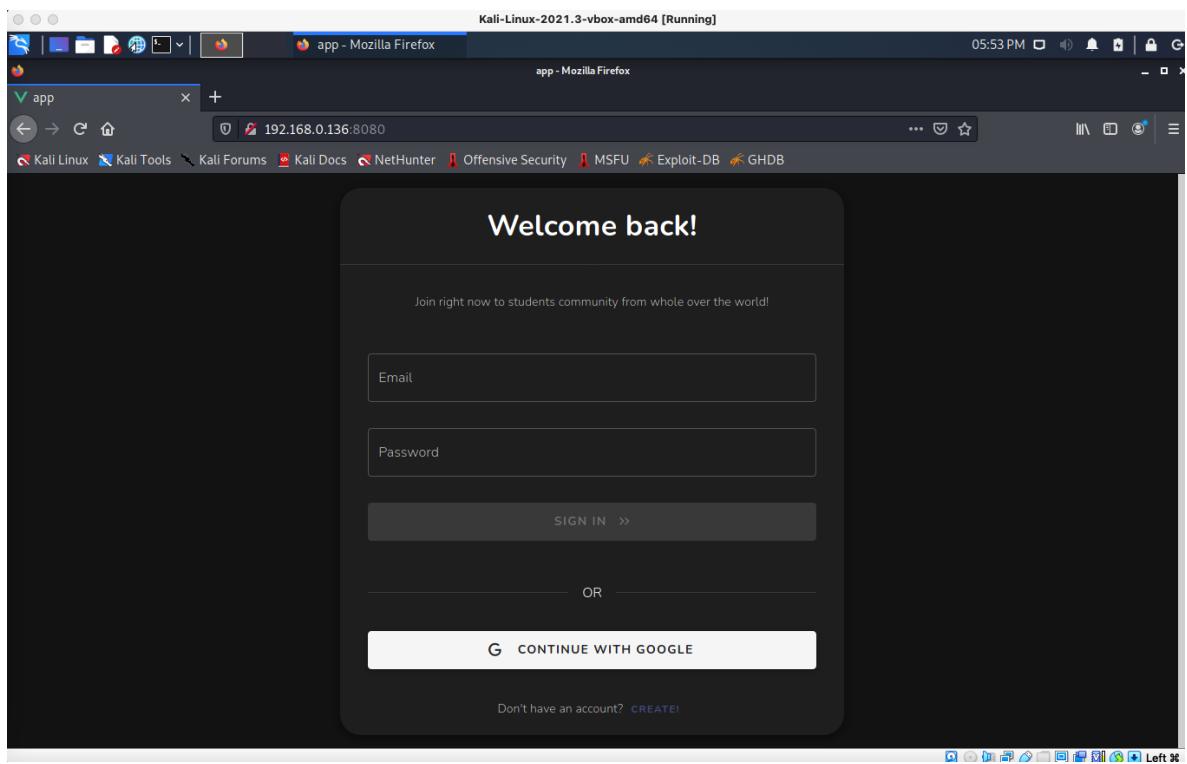
2 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
```

Rysunek 6.21 Naprawienie podatności na Regex DoS z pomocą odpowiedniego narzędzia

Jak się okazało, wystarczającym było użycie wyżej wspomnianej komendy do naprawy podatności na *Regex DoS*, otworzonych przez zależności zewnętrzne. Rysunek 6.21 przedstawia, że zaktualizowanie paczek do najnowszych oraz najbezpieczniejszych bezpośrednio rozwiązało problem.

Zabezpieczanie aplikacji przed logicznym atakiem *DoS* jest znacznie trudniejsze. Aby przetestować działanie takiego ataku na mojej aplikacji, użyłem do tego systemu *Kali Linux*.



Rysunek 6.22 Komunikator internetowy na systemie Kali Linux

Rysunek 6.22 przedstawia wystawienie aplikacji do lokalnej sieci. Zostało to zrobione po to, aby można było przetestować komunikator za pomocą odpowiednich narzędzi *Kali Linux*. Aplikacja działała pod adresem *IP* 192.168.0.136 oraz na porcie 8080. Pierwszą rzeczą, jaką chciałem sprawdzić była podatność serwera na zalanie pakietami *SYN*. W celu przygotowania wspomnianego ataku skorzystałem z *msfconsole* wbudowanego w *Kali Linux*, którego uruchomienie pokazałem na Rysunku 6.23. [22]

(kali㉿kali)-[~]

\$ msfconsole

192.168.0.136:8080

Welcome back

To boldly go where no shell has gone before

[ metasploit v6.1.4-dev ]

[ 2162 exploits - 1147 auxiliary - 367 post ]

[ 592 payloads - 45 encoders - 10 nops ]

[ 8 evasion ]

Metasploit tip: When in a module, use `back` to go back to the top level prompt

msf6 > [

Rysunek 6.23 Uruchomienie msfconsole

Na Rysunku 6.24 przedstawiłem konfigurację ataku na odpowiedni adres *IP* oraz port.

```
msf6 > use auxiliary/dos/tcp/synflood
msf6 auxiliary(dos/tcp/synflood) > show options

Module options (auxiliary/dos/tcp/synflood):
Name      Current Setting  Required  Description
INTERFACE          no        The name of the interface
NUM                no        Number of SYNs to send (else unlimited)
RHOSTS             yes       The target host(s), see https://github.com/rapi
                           sing-Metasploit
RPORT              80        yes       The target port
SHOST               no        The spoofable source address (else randomizes)
SNAPLEN            65535     yes       The number of bytes to capture
SPORT               no        The source port (else randomizes)
TIMEOUT            500       yes       The number of seconds to wait for new data

msf6 auxiliary(dos/tcp/synflood) > set RHOST 192.168.0.136
RHOST => 192.168.0.136
msf6 auxiliary(dos/tcp/synflood) > set RPORT 8080
RPORT => 8080
msf6 auxiliary(dos/tcp/synflood) > [
```

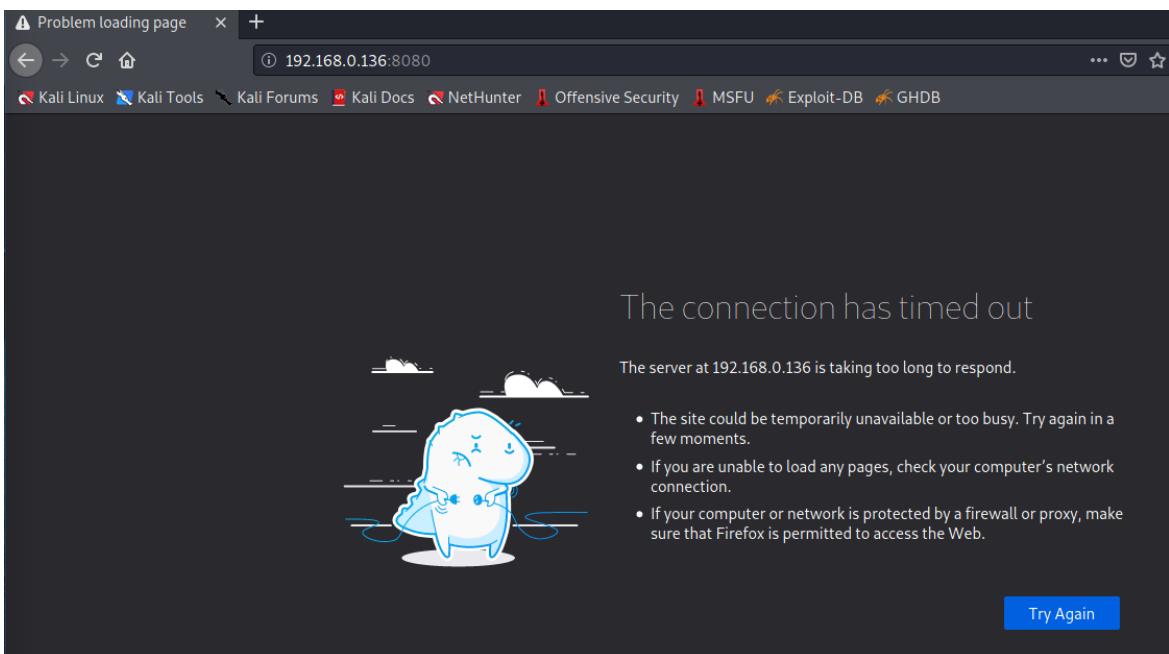
Rysunek 6.24 Konfiguracja ataku SYN flooding

Jak możemy zauważyc na Rysunku 6.25, atak polegający na zalewaniu ofiary pakietami SYN został uruchomiony pomyślnie.

No.	Time	Source	Destination	Protocol	Length	Info
36712	317.995972063	215.191.140.120	192.168.0.136	TCP	54	[TCP Port numbers reused]
36713	317.996587689	215.191.140.120	192.168.0.136	TCP	54	14072 → 8080 [SYN] Seq:
36714	317.997223772	215.191.140.120	192.168.0.136	TCP	54	49979 → 8080 [SYN] Seq:
36715	317.997896929	215.191.140.120	192.168.0.136	TCP	54	24015 → 8080 [SYN] Seq:
36716	317.998788735	215.191.140.120	192.168.0.136	TCP	54	[TCP Port numbers reused]
36717	317.999612194	215.191.140.120	192.168.0.136	TCP	54	[TCP Port numbers reused]
36718	318.000247492	215.191.140.120	192.168.0.136	TCP	54	62009 → 8080 [SYN] Seq:
36719	318.000906171	215.191.140.120	192.168.0.136	TCP	54	44698 → 8080 [SYN] Seq:
36720	318.001507472	215.191.140.120	192.168.0.136	TCP	54	[TCP Port numbers reused]
36721	318.002123791	215.191.140.120	192.168.0.136	TCP	54	51578 → 8080 [SYN] Seq:
36722	318.002753866	215.191.140.120	192.168.0.136	TCP	54	23873 → 8080 [SYN] Seq:

Rysunek 6.25 Rejestr z wireshark pokazujący wysyłane pakiety podczas SYN flooding

Po tak wykonanym ataku wejście na aplikację stało się niemożliwe – serwer został na tyle obficie zalany pakietami, że nie mógł sobie z nimi poradzić, czego skutek można zobaczyć na Rysunku 6.26.



Rysunek 6.26 Skutek ataku SYN flooding

Otrzymawszy taki wynik testu serwera na atak *DoS* polegający na zalewaniu ofiary pakietami *SYN* wiemy, że należałyby zainwestować w mechanizmy obronne przed takimi atakami, gdyż mogą one w łatwy sposób wyłączyć naszą aplikację z poprawnego działania. Aby przeciwdziałać takim atakom można na przykład zaimplementować filtrowanie pakietów na firewall'u albo też strategię przeciwdziałania zalewania serwera pakietami *SYN* za pomocą tak zwanych *SYN Cookies*. Wspomniane sposoby zabezpieczenia nieco odbiegają od tematyki oraz założeń mojej pracy, dlatego też nie zostaną one tutaj przedstawione.

Sposób, w jaki zabezpieczyłem aplikację przed nadmiernym żądaniem dostępu do zasobów – inaczej logiczny *DoS* – polega na zaimplementowaniu mechanizmu limitującego ilość żądań per IP użytkownika w określonej ramie czasowej. Implementacja mechanizmu w postaci kodu *JavaScript*, przedstawiona jest poniżej:

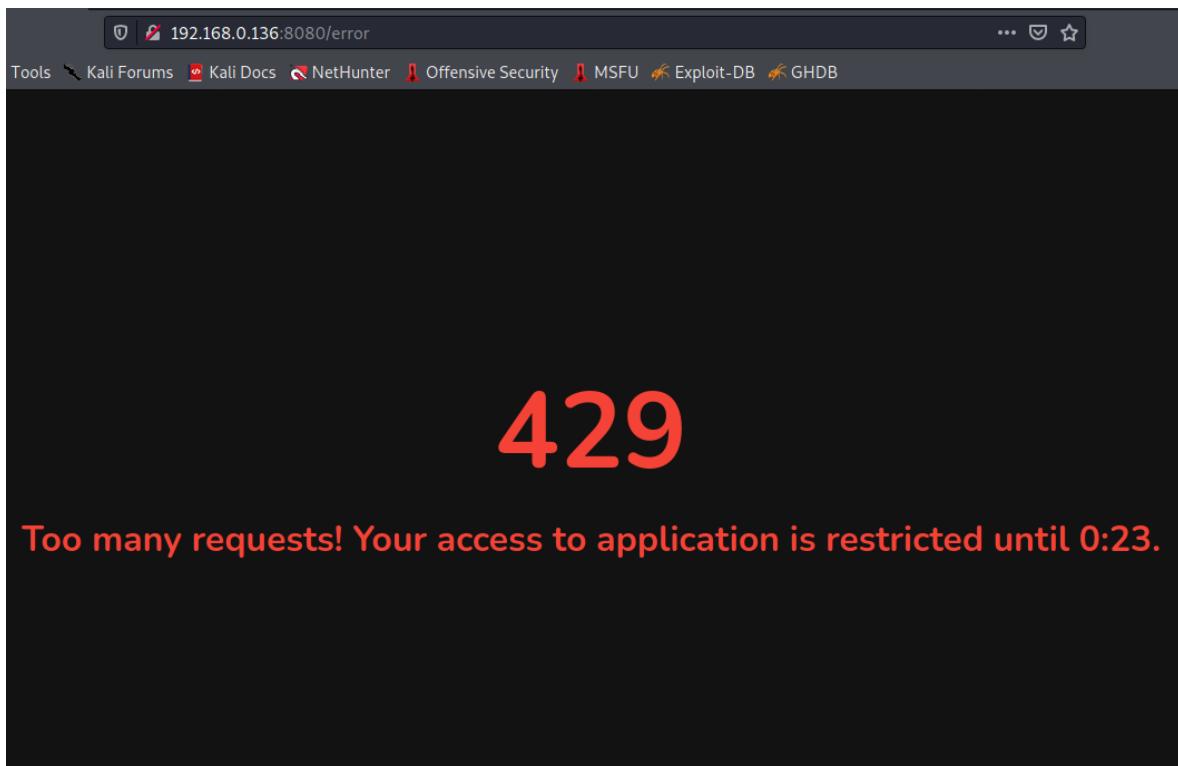
```
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100,
  message: "Too many requests! Your access to application is restricted until ",
  handler: (req, res, _, options) => {
    res.status(options.statusCode).send({
      message: `${options.message}${dayjs(req.rateLimit.resetTime).format(
        "H:mm"
      )}`,
    });
  },
});
app.use(limiter);
```

Rysunek 6.27 Mechanizm obronny przed logicznym DoS

Z Rysunku 6.27 można wyczytać, że mechanizm ten blokuje dostęp użytkownika, do jakichkolwiek zasobów aplikacji, na 15 minut, jeśli użytkownik przekroczy ilość 100 żądań w ciągu minuty. Dodatkowo, po zaimplementowaniu ww. mechanizmu, na Rysunku 6.28 widzimy, że do każdego żądania dodawane są nagłówki informujące nas o ustawionym limicie żądań, pozostałych dostępnych żądań w danej dziedzinie czasu oraz znacznik czasowy wskazujący nam, kiedy ilość dozwolonych żądań się zresetuje do wartości pierwotnej – 100;

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 98
X-RateLimit-Reset: 1639002781
```

Rysunek 6.28 Nagłówki z informacją dotyczącą ilości dostępnych żądań w zadanej dziedzinie czasu



Rysunek 6.29 Efekt działania mechanizmu obronnego przed logicznym DoS

Mechanizm skutecznie poradził sobie z zablokowaniem użytkownika, który przekroczył dozwoloną ilość żądań w określonym czasie, co jest przedstawione na Rysunku 6.29. W codziennym użytkowaniu bardzo mało prawdopodobne jest, aby w aplikacji typu komunikator internetowy, ktokolwiek wykonał więcej niż 100 żądań w ciągu minuty. Dlatego też zaimplementowanie takiego mechanizmu obronnego pozwoli na start odfiltrować oraz zapobiec potencjalnym atakom *DoS*.

Na koniec warto też sprawdzić zewnętrzne zależności testowanej aplikacji pod kątem bezpieczeństwa. Testowana aplikacja używa menedżera paczek zwanego *node package manager*, w skrócie – *npm*. *Npm* jest najbardziej powszechnym menedżerem paczek dla języka *JavaScript* – czyli tego, którego używam w aplikacji. *Npm* posiada wiele przydatnych komend, między innymi takie, które listują zewnętrzne zależności, używane przez aplikacje, w postaci drzewka, sprawdzają je pod kątem bezpieczeństwa przeprowadzając audyt oraz ewentualnie naprawiają znalezione luki, najczęściej poprzez aktualizację zależności do najnowszych, czyli potencjalnie bezpieczniejszych wersji (zawierającymi zabezpieczenia przed wcześniej znalezionymi lukami bezpieczeństwa). Na początek wykonam wylistowanie zależności w postaci drzewka za pomocą komendy *npm ls*, co przedstawiłem na Rysunku 6.30.

```
→ app-api git:(main) npm ls
app-api@1.0.0 /Users/rafonix/video-chat/app-api
├── @socket.io/redis-adapter@7.0.0
├── bcrypt@5.0.1
├── bluebird@3.7.2
├── cookie-parser@1.4.5
├── cookie-session@1.4.0
├── cors@2.8.5
├── dayjs@1.10.7
├── dotenv@10.0.0
├── express-rate-limit@5.5.0
├── express-validator@6.13.0
├── express@4.17.1
├── http@0.0.1-security
├── jsonwebtoken@8.5.1
├── mysql2@2.3.0
├── nodemon@2.0.13
├── passport-google-oauth2@2.0.0
├── passport@0.5.0
├── path@0.12.7
├── redis@3.1.2
├── sequelize-cli@6.2.0
├── sequelize@6.6.5
└── socket.io@4.1.3
```

Rysunek 6.30 Drzewko zewnętrznych zależności serwera API

Następnie wykonam audyt znalezionych zewnętrznych zależności, w celu wyszukania potencjalnych podatności z nimi związanych. Komenda, która wykona wyżej wspomniany audyt, to *npm audit*.

```
37 vulnerabilities (23 moderate, 14 high)

To address issues that do not require attention, run:
  npm audit fix

To address all issues possible (including breaking changes), run:
  npm audit fix --force

Some issues need review, and may require choosing
a different dependency.
```

Rysunek 6.31 Raport z audytu zależności serwera API

Przedstawiony na Rysunku 6.31 audyt znalazł łącznie 37 podatności spowodowanych zależnościami zewnętrznymi, w tym 23 o poziomie umiarkowanym i aż 14 podatności wysokiego poziomu – czyli takich, które niezwłocznie trzeba usunąć, gdyż stanowią niebezpieczeństwo dla aplikacji wystawiając, wspomniane w audycie, luki bezpieczeństwa.

W takiej sytuacji, najlepszym pomysłem jest jak najszybsze pozbycie się problemów. Można tego dokonać, jak już *npm* nam podpowiada, za pomocą komendy *npm audit fix*, jeśli chcemy naprawić tylko te zależności, które są kompatybilne z obecnie używanymi wersjami paczek albo za pomocą *npm audit fix -force*, jeśli chcemy naprawić dosłownie wszystkie luki bezpieczeństwa. Druga opcja może na pierwszy rzut oka brzmieć bardziej atrakcyjnie, aczkolwiek pobierze ona bezwzględnie najnowsze wersje zależności – nawet jeśli posiadają one przełomowe zmiany. Może to spowodować zepsucie działania aplikacji, a więc ogromną stratę czasu w próbie naprawienia jej.

```
→ app-api git:(main) npm audit fix --dry-run
added 1 package, removed 12 packages, changed 17 packages, and audited 369 packages
in 2s

23 packages are looking for funding
  run 'npm fund' for details
```

Rysunek 6.32 Próba dokonania naprawy wcześniej znalezionych luk bezpieczeństwa

Jak widać na Rysunku 6.32, zdecydowałem się na pierwszą opcję, czyli *npm audit fix*, ponieważ uznałem, że nie chce bezwzględnie aktualizować wszystkich zależności, a następnie trudzić się prawdopodobnie naprawą aplikacji. Do komendy została dodana też flaga *-dry-run*, która powoduje jedynie wygenerowanie raportu zawierającego akcje, które podejmie komenda *npm audit fix*, nie dokonując tak naprawdę żadnych zmian.



Rysunek 6.33 Wynik końcowy audytu wraz z możliwymi przeprowadzonymi naprawami podatności

Użyta komenda okazała się być wystarczająca, ponieważ naprawiła większość znalezionych podatności oraz błędów nie psując przy tym działania aplikacji, czemu dowodzi Rysunek 6.33. Dziewięć podatności o umiarkowanym stopniu nie powinno wpływać na zmianę zdania osoby audytującej o użyciu komendy *npm audit fix -force*, gdyż luki takie nie są szczególnie niebezpieczne.

### 6.3. Analiza wyników i dalsze działania

Szereg testów wykonanych na aplikację pozwolił ocenić, w jakim stopniu jest ona zabezpieczona oraz w jakich miejscach występowały potencjalne luki bezpieczeństwa. Jak możemy zauważyć, aplikacja, jeszcze przed jej dokładniejszym zabezpieczeniem, była już stosunkowo bezpieczna. Wynika to z tego, że została ona zbudowana w nowoczesnych technologiach, które z reguły wymuszają nieco na programiście budowanie jej w sposób bezpieczny. Dodatkowo, biblioteki oraz inne zewnętrzne zależności były, już przed ich użyciem, starannie wybierane pod kątem ich bezpieczeństwa.

Testy pozwoliły wykryć podatności aplikacji na wspomniane w pracy rodzaje ataków XSS. Stosując zawarte w części teoretycznej sposoby na zabezpieczenie przed wspomnianymi atakami, udało się w znacznej mierze zapobiec tym podatnościom, a dodatkowe zimplementowanie *CSP* niemalże zniwelowało możliwości wykonywania tych ataków.

W przypadku *CSRF* zabezpieczenie okazało się być stosunkowo proste w implementacji oraz skuteczne w działaniu. W dużej mierze pomogło mi narzędzie *Postman*, dzięki któremu bez większego wysiłku mogłem wykonywać żądania do mojej aplikacji „z zewnątrz”. Zaimplementowanie mechanizmu *CSRF Token* pozwoliło wyeliminować lukę związaną z możliwością przeprowadzania ataków z zewnątrz za pośrednictwem osób zautoryzowanych. Jedyną możliwością ataku jest wykradnięcie ważnego tokenu *CSRF*, co znacznie ogranicza możliwości atakującego, gdyż, aby przeprowadzić atak na dużą skalę, musiałby ukraść tokeny tysiącom użytkowników co jest nie lada wyzwaniem.

Faktem jest też, że aplikacja od momentu projektowania była już zabezpieczona przed atakiem *SQL Injection*. Pomogło w tym zastosowanie nowoczesnego frameworku oraz biblioteki do mapowania obiektowo-relacyjnego podczas przeprowadzania jakichkolwiek zapytań do bazy danych. Mapowanie takie stosuje np. spараметyzowane zapytania co zapobiega wykonywaniu się omawianego typu ataku.

Zabezpieczenie przed atakiem typu *Regex DoS* było stosunkowo proste, a pomogły w tym metody zaprezentowane w teoretycznej części pracy. Po odpowiednim przeskanowaniu aplikacji, a następnie naprawieniu wykrytych błędów, podatność została zneutralizowana. Znacznie większym wyzwaniem okazał się logiczny *DoS*. Udało mi się zablokować nadmierną ilość żądań użytkownika do serwera w określonej dziedzinie czasu, jednakże nadal możliwe jest zalewanie aplikacji pakietami *SYN* (*SYN flood*), przez co wykonanie takiego ataku przez hakera może całkowicie pozbawić naszą aplikację prawidłowego działania.

Na koniec został wykonany skan zależności zewnętrznych w poszukiwaniu potencjalnych luk bezpieczeństwa wynikających z użycia tych właśnie zależności. Skan wykazał jedynie 14 poważniejszych podatności, które można było w łatwy sposób zniwelować aktualizując odpowiednie paczki z użyciem narzędzia *npm*.

Po przeprowadzeniu testów oraz wykonaniu dodatkowych zabezpieczeń podatności wykrytych przez testy, stan bezpieczeństwa aplikacji w dość dużym stopniu uległ poprawie. Nadal istnieją poważne podatności na np. *SYN flooding*, które należałyby niezwłocznie usunąć w przypadku sytuacji wystawienia aplikacji do sieci. Należałyby też niezwłocznie przeskanować aplikację komercyjnymi skanerami, ponieważ one tak naprawdę pokazałyby wszystkie luki bezpieczeństwa oraz przeprowadziłyby testy ataków, które nie zostały zawarte w niniejszej pracy. Następnie, na podstawie uzyskanych wyników ze skanera, należałyby dokonać kolejnych, sugerowanych zabezpieczeń. Na stronie *OWASP* [4] można znaleźć zestawienie najgroźniejszych ataków na aplikacje internetowe oraz rekomendacje w zabezpieczaniu przed nimi, co jest na pewno dobrym krokiem w dalszym zabezpieczaniu aplikacji.

## 7. Podsumowanie

Zaprojektowanie aplikacji typu komunikator internetowy oraz wdrożenie go za pomocą maszyn wirtualnych w celu badań podatności przebiegło pomyślnie. Wdrożenie aplikacji jako stanowiska do badań podatności zostało przeprowadzone za pomocą dwóch maszyn wirtualnych w odpowiadających stanach: przed oraz po zabezpieczeniu. Takie stanowisko pozwala na dalsze przeprowadzanie badań jego bezpieczeństwa na różne podatności, a ponadto umożliwia porównanie stopnia bezpieczeństwa aplikacji w dwóch ww. stanach.

Niniejsza praca w zwięzły, ale i wyczerpujący, sposób przedstawiła temat bezpieczeństwa nowoczesnych aplikacji internetowych. Stopień ich bezpieczeństwa znacznie różni się w stosunku do aplikacji budowanych jeszcze dekadę temu, ponieważ, w dzisiejszych czasach, kładzie się jeszcze większy nacisk na zabezpieczenia, a ponadto wielu powszechnym oraz niebezpiecznym w przeszłości atakom udało się zapobiec. Aby jednak bezpieczeństwo aplikacji miało przełożenie w rzeczywistości, należy odpowiednio przemyśleć jej architekturę oraz zaprojektować ją w sposób bezpieczny jeszcze przed budowaniem (programowaniem) aplikacji. Takie podejście pozwoli uniknąć wiele błędów w przyszłości, które wówczas będą bardzo trudne albo niemożliwe do naprawienia.

Poza umiejętnością zaprojektowania odpowiedniej architektury aplikacji, zastosowaniu dobrych praktyk oraz doboru odpowiednich narzędzi oraz mechanizmów (np., uwierzytelniania) ważna jest też znajomość różnych rodzajów ataków na aplikacje internetowe oraz sposobów zabezpieczania przed nimi. W pracy zostały przedstawione jedne z najpopularniejszych ataków na aplikacje internetowe, a ponadto zaprezentowane zostały sposoby ich wykorzystania, a także praktyczne przykłady mechanizmów obronnych. Należy odpowiednio doszkalać programistów w tym zakresie, ponieważ większość przedstawionych w pracy ataków niesie za sobą bardzo poważne skutki, którym znacznie łatwiej i taniej zapobiegać niż naprawiać po fakcie.

Na koniec pracy, czyli w jej części praktycznej, przeprowadzony został szereg testów na komunikatorze internetowym. Celem tych testów było zweryfikowanie założenia, przedstawionego we wstępie do pracy - czy nowoczesne aplikacje internetowe są z reguły bezpieczne. Wynikiem przeprowadzonych testów w warunkach lokalnych oraz za pomocą maszyny wirtualnej był fakt, że na niektóre z przedstawionych ataków aplikacja jest domyślnie odporna i zabezpieczona jeszcze przed wprowadzeniem jakichkolwiek własnych mechanizmów bezpieczeństwa. W przypadku wykrycia podatności, od razu zostały podjęte próby im zapobiegnięcia. Dobrze zaprojektowana architektura aplikacji pozwoliła na w dość łatwy i bez wysiłkowy sposób „załatwania” luk bezpieczeństwa. Jedyne pojedyncze z wykrytych podatności nie zostały do końca załatwane, np. *SYN flood* – głównie dlatego, że sposób zapobiegania wspomnianym atakom wykraczałby poza zakres tematyczny poruszany w pracy.

Otrzymane wyniki końcowe, po przeprowadzeniu testów i zaimplementowaniu mechanizmów bezpieczeństwa, jednoznacznie określiły, że komunikator internetowy budowany zgodnie z nowoczesnymi, dobrymi praktykami, można uznać za bezpieczny szczególnie w porównaniu do nieprawidłowo zaprojektowanej aplikacji. Nawet jeśli zostanie wykryta jakakolwiek podatność w przyszłości, dobrze przemyślana oraz zaprojektowana architektura aplikacji pozwoli w łatwy oraz szybki sposób pozbyć się danej luki bezpieczeństwa.

Ponadto stworzony na potrzeby pracy komunikator internetowy jest w idealnej fazie do dalszego rozwoju. Można by było ulepszyć mechanizm wideo rozmów oraz dodać możliwość tworzenie wideo konferencji tak aby aplikacja mogłaby być jeszcze szerzej wykorzystywana – nie tylko w celach prywatnych, ale i biznesowych. Ponadto dobrym pomysłem byłoby dodanie możliwości wrzucania plików czy obrazków do rozmowy tak aby

móc się nimi podzielić z innym uczestnikami konwersacji. Mogłaby być to świetna alternatywa dla okrojonych już powszechnie dostępnych komunikatorów.

## Literatura

- [1] Andrew Hoffman, *Web Application Security*, O'Reilly 2020
- [2] Bentkowski M., *Podatność Cross-Site Scripting (XSS)*, w: *Bezpieczeństwo aplikacji webowych*, Michał Bentkowski, Gynvael Coldwind i inni, Securitum 2019
- [3] John Mitchell, *Web Application Security*,  
<https://crypto.stanford.edu/cs155old/cs155-spring11/lectures/10-web-site-sec.pdf>  
[dostęp: 24.10.2021]
- [4] *Top 10 Web Application Security Risks*, <https://owasp.org/Top10/> [dostęp: 24.10.2021]
- [5] *Vue.js*, <https://vuejs.org/v2/guide/> [dostęp: 24.10.2021]
- [6] Przemysław Bykowski, *REST API – Efektywna Droga Do Zrozumienia*,  
<https://bykowski.pl/rest-api-efektywna-droga-do-zrozumienia/> [dostęp: 25.10.2021]
- [7] Michał Sajdak, *(Nie)bezpieczeństwa JWT (JSON Web Token)*,  
<https://sekurak.pl/jwt-security-ebook.pdf> [dostęp: 26.10.2021]
- [8] Flavio Copes, *JWT authentication: Best practices and when to use it*,  
<https://blog.logrocket.com/jwt-authentication-best-practices/> [dostęp: 27.10.2021]
- [9] Marcin Piasek, *OAUTH .20 – jak działa / jak testować / problemy bezpieczeństwa*,  
<https://sekurak.pl/oauth-2-0-jak-dziala-jak-testowac-problemy-bezpieczenstwa/>  
[dostęp: 27.10.2021]
- [10] *Czym jest DOM?*,  
[https://developer.mozilla.org/pl/docs/Web/API/Document\\_Object\\_Model/Introduction](https://developer.mozilla.org/pl/docs/Web/API/Document_Object_Model/Introduction) [dostęp: 10.11.2021]
- [11] *npm Docs*, <https://docs.npmjs.com/> [dostęp: 12.11.2021]
- [12] *Content Security Policy (CSP)*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP> [dostęp: 15.11.2021]
- [13] *Using the Fork-and-Branch Git Workflow*,  
<https://blog.scottlowe.org/2015/01/27/using-fork-branch-git-workflow/> [dostęp: 16.11.]
- [14] *Common Vulnerabilities and Exposures database*, <https://cve.mitre.org/index.html>  
[dostęp: 16.11.2021]
- [15] *National Vulnerability Database*, <https://nvd.nist.gov/> [dostęp: 16.11.2021]

- [16] John Au-Yeung, *Prevent Cross-Site Request Forgery in Express Apps with csurf*, <https://medium.com/dataseries/prevent-cross-site-request-forgery-in-express-apps-with-csurf-16025a980457> [dostęp: 24.11.2021]
- [17] *MySQL Documentation*, <https://dev.mysql.com/doc/> [dostęp: 25.11.2021]
- [18] *Express morgan middleware*,  
<https://expressjs.com/en/resources/middleware/morgan.html> [dostęp: 02.12.2021]
- [19] *Node.js v15.14.0 documentation*, <https://nodejs.org/docs/latest-v15.x/api/> [dostęp: 06.12.2021]
- [20] *Production Best Practices: Security*, <https://expressjs.com/en/advanced/best-practice-security.html> [dostęp: 06.12.2021]
- [21] *Sequelize*, <https://sequelize.org/master/> [dostęp: 09.10.2021]
- [22] Joseph Munitz, Aamir Lakhani, *Kali Linux testy penetracyjne*, Helion, 2014