

# Zarządzanie infrastrukturą teleinformatyczną

## PROJEKT

Temat projektu:

Implementacja serwisu webowego podatnego na ataki OWASP Top Ten

Prowadzący:

-----

Grupa zajęciowa:

-----

Autor projektu:

Kacper Hanuszewicz

Data oddania projektu:

2 lutego 2022 r.

Politechnika Wrocławska

Wrocław 2022

## Spis treści

1. Wstęp, cele projektu
2. Wykorzystane technologie
3. Opis aplikacji
4. Omówienie działania aplikacji
5. Prezentacja wybranych luk OWASP Top Ten na przykładzie napisanej aplikacji
6. Podsumowanie

### 1. Wstęp, cele projektu

Realizując grupę kursów Zarządzanie infrastrukturą teleinformatyczną należało wykonać projekt. Spośród dostępnych tematów wybrano temat zatytułowany „Implementacja serwisu webowego podatnego na ataki OWASP Top Ten”. Celem tego projektu było napisanie aplikacji sieciowej podatnej na co najmniej 5 luk bezpieczeństwa.

### 2. Wykorzystane technologie

Wybrano język programowania JavaScript. Został on użyty do napisania kodu serwerowego, wykorzystując przy tym środowisko uruchomieniowe Node.js oraz framework Express.js. Wykorzystana baza danych to MongoDB. Użyte środowisko programistyczne to Visual Studio Code. Został także wykorzystany menadżer pakietów npm.

### 3. Opis aplikacji

Wykonana aplikacja to kod serwerowy REST API. Program nie posiada części graficznej frontend. Motywowane jest to faktem, iż stworzenie i prezentacja wybranych luk bezpieczeństwa nie wymaga interfejsu graficznego z poziomu przeglądarki. Do wysyłania żądań i odbierania odpowiedzi został użyty program Postman komunikujący się z serwerem za pomocą plików JSON.

Aplikacja jest blogiem, który służy tworzeniu i przeglądaniu postów. Przed postowaniem należy utworzyć konto i się zalogować. Dostępne funkcjonalności obejmują aktualizowanie danych użytkownika, usuwanie użytkowników, aktualizowanie postów oraz usuwanie postów. Dane są przechowywane w zdalnym klastrze bazy danych MongoDB.

Za solenie i haszowanie haseł odpowiada moduł bcrypt. Zaimplementowany został prosty mechanizm logowania.

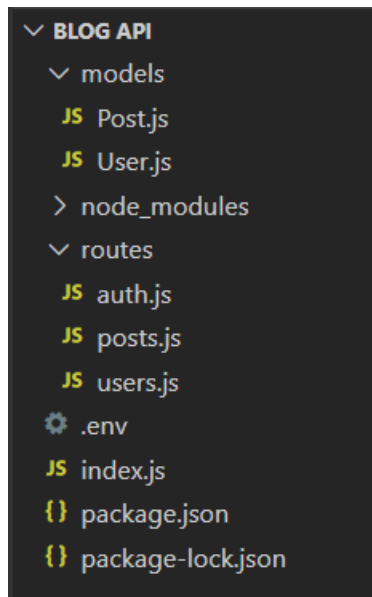
#### 4. Omówienie działania aplikacji

```
{ } package.json > ...
1  {
2    "dependencies": {
3      "bcrypt": "^5.0.1",
4      "dotenv": "^15.0.0",
5      "express": "^4.17.2",
6      "mongoose": "^6.1.9"
7    },
8    "name": "blog-api",
9    "version": "1.0.0",
10   "description": "Aplikacja na projekt ZiT",
11   "main": "index.js",
12   "devDependencies": {
13     "nodemon": "^2.0.15"
14   },
15   "scripts": {
16     "start": "nodemon index.js"
17   },
18   "author": "Kacper Hanuszewicz",
19   "license": "MIT"
20 }
21
```

*Obraz 1 – plik package.json*

Zostały użyte takie moduły jak:

- bcrypt, odpowiadający za solenie i haszowanie haseł
- dotenv, służący do zarządzania zmiennymi środowiskowymi
- express, czyli framework do Node.js, na którym opiera się backendowe działanie całej aplikacji
- mongoose, pozwalający na modelowanie obiektów MongoDB i pracę w środowisku asynchronicznym
- nodemon, dzięki któremu serwer się automatycznie restartował po każdym zapisaniu pliku z projektu



Obraz 2 – struktura katalogowa aplikacji

```
const UserSchema = new mongoose.Schema({
  {
    username: {
      type: String,
      required: true,
      unique: true,
    },
    email: {
      type: String,
      required: true,
      unique: true,
    },
    password: {
      type: String,
      required: true,
    },
  },
  { timestamps: true }
});

const PostSchema = new mongoose.Schema({
  {
    title: {
      type: String,
      required: true,
      unique: true,
    },
    desc: {
      type: String,
      required: true,
    },
    username: {
      type: String,
      required: true,
    },
  },
  { timestamps: true }
});
```

Obraz 3 – schematy tworzenia użytkownika i postu (User.js oraz Post.js)

Jako bazowy plik JavaScript użyto index.js. Katalog routes zawiera części kodu odpowiadające za, jak sama nazwa wskazuje, ścieżki w aplikacji sieciowej. Plik auth.js zawiera obsługę rejestracji (ścieżka /register) oraz logowania (ścieżka /login).

```
// rejestracja
router.post("/register", async (req, res) => {
  try {
    const salt = await bcrypt.genSalt(1);
    const hashedPass = await bcrypt.hash(req.body.password, salt);
    const newUser = new User({
      username: req.body.username,
      email: req.body.email,
      password: hashedPass,
    });

    const user = await newUser.save();
    res.status(200).json(user);
  } catch (err) {
    res.status(500).json(err);
  }
});
```

*Obraz 4 – kod obsługujący rejestrację nowego użytkownika (auth.js)*

Użyto składni async/await ze względu na wymagania mongoose, jednak przy niewielkim projekcie nie ma to większego znaczenia.

Użyta metoda to post. Za pomocą bcrypt hasła zostają solone (słabo, zostanie to rozwinięte podczas omawiania luk) i haszowane, a korzystając z metod mongoose tworzony jest nowy użytkownik. Serwer zwraca odpowiednie kody odpowiedzi http.

```

// logowanie
router.post("/login", async (req, res) => {
  try {
    const user = await User.findOne({ username: req.body.username });
    !user && res.status(400).json("Wrong credentials!");

    const validated = await bcrypt.compare(req.body.password, user.password);
    !validated && res.status(400).json("Wrong credentials!");

    //const { password, ...others } = user._doc;
    res.status(200).json(user);
  } catch (err) {
    res.status(500).json(err);
  }
});

```

*Obraz 5 – kod obsługujący logowanie użytkownika (auth.js)*

Użyta metoda to post. Mongoose szuka w bazie danych nazwy użytkownika, bcrypt porównuje wpisane hasło z hasłem w bazie danych. Serwer zwraca odpowiednie kody odpowiedzi http.

```

// get user
router.get("/:id", async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    const { password, ...others } = user._doc;
    res.status(200).json(others);
  } catch (err) {
    res.status(500).json(err);
  }
});

```

*Obraz 6 – kod odpowiadający za żądanie danych użytkownika (users.js)*

Plik users.js zawiera fragmenty kodu get user, update user oraz delete user. Wykorzystuje również mongoose, a także wysyła kody odpowiedzi HTTP.

```

// delete post
router.delete("/:id", async (req, res) => {
  try {
    const post = await Post.findById(req.params.id);
    if (post.username) { // przypadkiem usunięto: === req.body.username
      try {
        await post.delete();
        res.status(200).json("Post has been deleted...");
      } catch (err) {
        res.status(500).json(err);
      }
    } else {
      res.status(401).json("You can delete only your post!");
    }
  } catch (err) {
    res.status(500).json(err);
  }
});

```

*Obraz 7 – kod odpowiadający za usuwanie postów (posts.js)*

Plik posts.js zawiera fragmenty kodu get post, create post, delete post oraz update post. Wykorzystuje również mongoose, wysyła kody odpowiedzi HTTP, jednak nie weryfikuje tożsamości użytkownika, co zostanie rozwinięte przy omówieniu luk.

```

const express = require("express");
const dotenv = require("dotenv");
const mongoose = require("mongoose");
const authRoute = require("./routes/auth");
const userRoute = require("./routes/users");
const postRoute = require("./routes/posts");

const app = express();

dotenv.config();
app.use(express.json());

mongoose
  .connect(process.env.MONGO_URL, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(console.log("Connected to MongoDB"))
  .catch((err) => console.log(err));

app.use("/auth", authRoute);
app.use("/users", userRoute);
app.use("/posts", postRoute);

app.listen("5000", () => {
  console.log("Server is running on port 5000...");
});

```

*Obraz 8 – rdzeń programu (index.js)*

Na początku importowane są odpowiednie moduły i pliki obsługujące ścieżki. Następnie tworzona jest instancja aplikacji. Potem plik .env ładowany jest do process.env. Aplikacji umożliwia się używanie plików JSON. Następuje połączenie z bazą danych mongoose (plik .env zawiera tylko adres do bazy danych przypisany do zmiennej środowiskowej MONGO\_URL). W dalszej części ustanawia się ścieżki w aplikacji. Krok ostatni polega na włączeniu nasłuchu serwera, aplikacja jest już gotowa do użytku.



## 5. Prezentacja wybranych luk OWASP Top Ten na przykładzie napisanej aplikacji

Wybrano pięć luk bezpieczeństwa.

### A01:2021 – Broken Access Control

Kontrola dostępu egzekwuje politykę w taki sposób, aby użytkownicy nie mogli działać poza zakresem swoich uprawnień. Niepożądane działania prowadzą zazwyczaj do nieuprawnionego ujawnienia informacji, modyfikacji lub zniszczenia danych lub wykonywania funkcji poza ograniczeniami użytkownika.

W aplikacji taka luka znajduje się w kodzie programu:



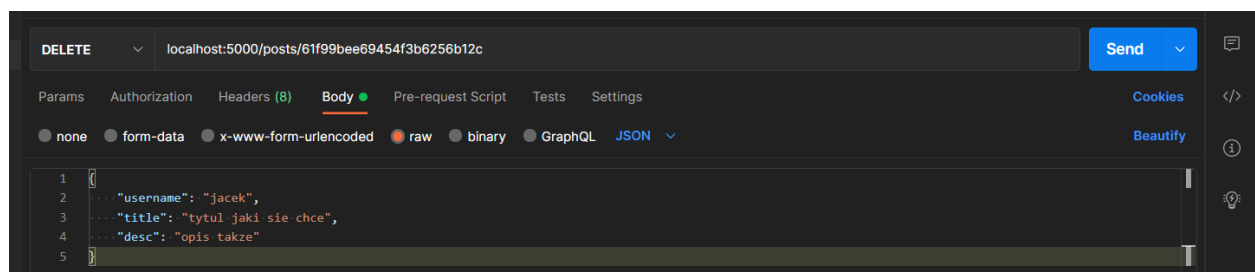
```
// delete post
router.delete('/:id', async (req, res) => {
  try {
    ! const post = await Post.findById(req.params.id);
    if (post.username) { // przypadkiem usunięto: === req.body.username
      try {
        await post.delete();
        res.status(200).json('Post has been deleted...');
      } catch (err) {
        res.status(500).json(err);
      }
    } else {
      res.status(401).json('You can delete only your post!');
    }
  } catch (err) {
    res.status(500).json(err);
  }
});
```

Obraz 9 – brak weryfikacji tożsamości użytkownika (auth.js)

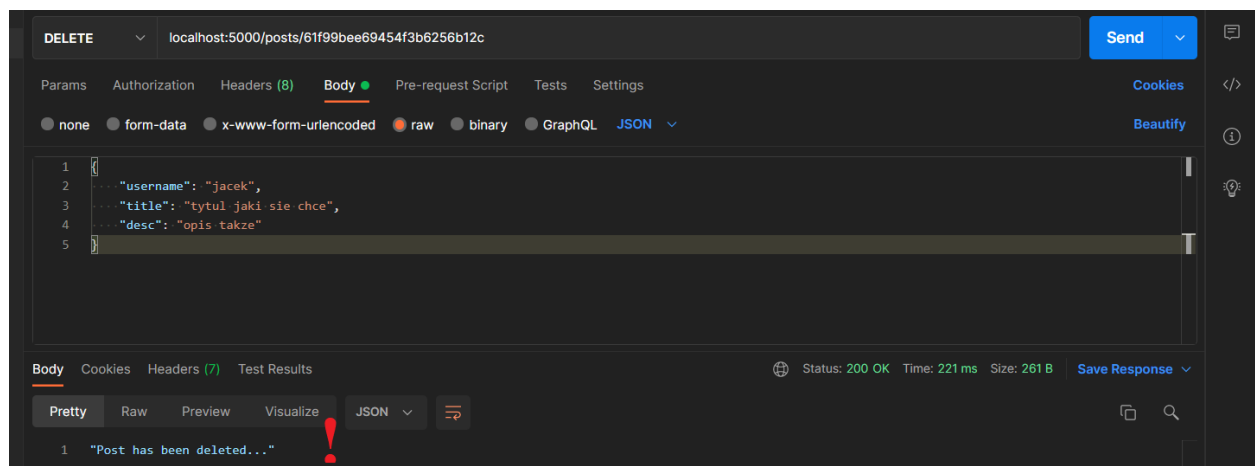
Warunek `post.username === req.body.username` został przypadkiem usunięty lub nie został w ogóle zaimplementowany. Skutkuje to tym, że dowolny użytkownik jest w stanie usunąć post dowolnego innego użytkownika. Jest to poważna luka, która, chociażby w aplikacjach bankowych, nie ma prawa mieć miejsca, ponieważ grozi to utratą poufnych danych na skalę krajową bądź globalną. W tej niewielkiej aplikacji można to naprawić w bardzo krótkim czasie przy niemal zerowym nakładzie pracy, jednak w większych aplikacjach szukanie takiego błędu może zająć bardzo wiele czasu i kosztować instytucję wiele zasobów, w międzyczasie narażając się na kolejne ataki wykorzystujące omawianą podatność.

```
  _id: ObjectId("61f99bee69454f3b6256b12c")
  title: "tytuł jaki sie chce"
  desc: "opis takze"
  username: "brucek"
> categories: Array
  createdAt: 2022-02-01T20:45:34.240+00:00
  updatedAt: 2022-02-01T20:45:34.240+00:00
  __v: 0
```

*Obraz 10 – post w bazie danych MongoDB*



*Obraz 11 – delete request od użytkownika jacek*



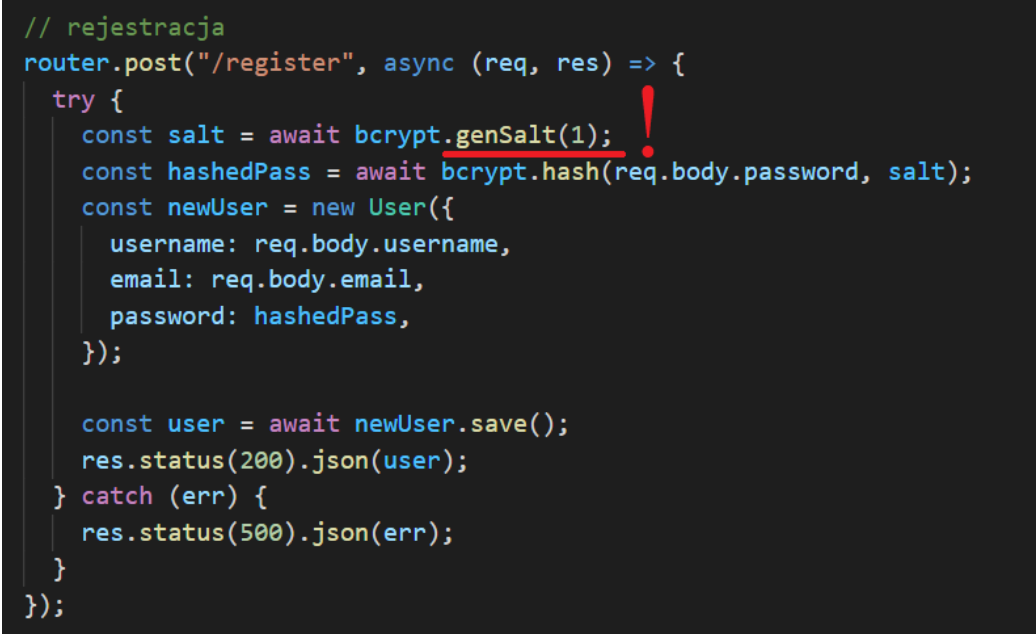
*Obraz 12 – komunikat o usunięciu postu użytkownika bruce przez użytkownika Jacek*

Post został usunięty przez użytkownika, który nawet nie istnieje w bazie danych. Jeszcze zmniejsza to bezpieczeństwo i eksponuje omawianą podatność niedostatecznej kontroli dostępu.

## A02:2021 – Cryptographic Failures

Niektóre dane wymagają zabezpieczenia w postaci zmiany plain textu na jakąś formę szyfrowania. Dotyczy to głównie przechowywanych w bazach danych haseł, numerów kart kredytowych, dokumentacji zdrowotnej, tajemnic handlowych czy innych danych mogących w jakiś sposób zidentyfikować osobę i/lub narazić konto w serwisie webowym na nieupoważniony dostęp.

W aplikacji taka luka znajduje się w kodzie programu:



```
// rejestracja
router.post("/register", async (req, res) => {
  try {
    const salt = await bcrypt.genSalt(1);
    const hashedPass = await bcrypt.hash(req.body.password, salt);
    const newUser = new User({
      username: req.body.username,
      email: req.body.email,
      password: hashedPass,
    });

    const user = await newUser.save();
    res.status(200).json(user);
  } catch (err) {
    res.status(500).json(err);
  }
});
```

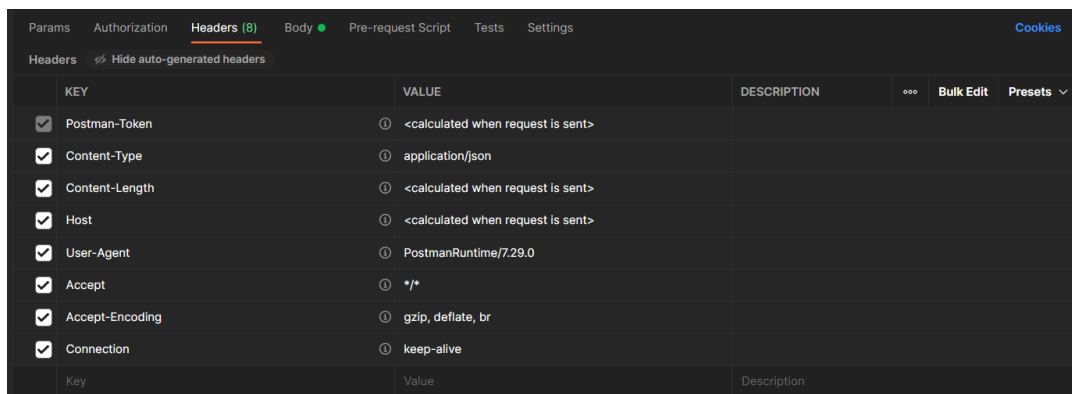
Obraz 13 – algorytm generowania soli i haszowania

Argument przekazany do funkcji odpowiada za siłę algorytmu. Im większy, tym lepszy, jednakże wraz z jego wzrostem zwiększa się wymagana moc obliczeniowa. Zalecaną wartością jest 10, co jest dobrym kompromisem pomiędzy siłą algorytmu oraz wymaganą mocą obliczeniową. W tym przypadku użyto najmniejszej dostępnej wartości, co sprawia, że szansa na złamanie hasła gwałtownie wzrasta. Jest to błąd, który może narazić konto użytkownika na nieuprawiony dostęp, a co za tym idzie, kradzież poufnych danych, danych osobistych, modyfikację konta czy nawet jego usunięcie bez zgody i wiedzy właściciela. W większych aplikacjach, szczególnie tych operujących na danych wrażliwych czy biznesowych, taka luka stanowi szczególne niebezpieczeństwo i nie powinna mieć prawa wystąpienia w kodzie.

## A05:2021 – Security Misconfiguration

Jest to zbiór podatności polegający na braku odpowiedniego wzmocnienia zabezpieczeń w dowolnej części stosu aplikacji lub nieprawidłowo skonfigurowanych uprawnieniach, włączeniu lub zainstalowaniu niepotrzebnych funkcji, istnieniu domyślnych kont z domyślnymi hasłami, ustawieniu zabezpieczeń na serwerach aplikacji, frameworkach aplikacji, bibliotekach, bazach danych itp. czy na niewysyłaniu przez serwer nagłówków ani dyrektyw bezpieczeństwa, ewentualnie nie są one ustawione na bezpieczne wartości.

Nagłówki przesyłane pomiędzy serwerem a użytkownikiem:



KEY	VALUE	DESCRIPTION		Bulk Edit	Presets
<input checked="" type="checkbox"/> Postman-Token	<calculated when request is sent>				
<input checked="" type="checkbox"/> Content-Type	application/json				
<input checked="" type="checkbox"/> Content-Length	<calculated when request is sent>				
<input checked="" type="checkbox"/> Host	<calculated when request is sent>				
<input checked="" type="checkbox"/> User-Agent	PostmanRuntime/7.29.0				
<input checked="" type="checkbox"/> Accept	*				
<input checked="" type="checkbox"/> Accept-Encoding	gzip, deflate, br				
<input checked="" type="checkbox"/> Connection	keep-alive				

Obraz 14 – widok na nagłówki w programie Postman

Serwer nie wysyła nagłówków bezpieczeństwa do klientów. Atakujący może skorzystać z faktu, że np. flaga HttpOnly jest domyślnie ustawiona na false i odczytać zawartość ciasteczka za pomocą JavaScript. Brak również nagłówka Content Security Policy, który jest skutecznym środkiem ochrony witryny przed atakami XSS. Poprzez whitelisting źródeł zatwierdzonych treści, można zapobiec ładowaniu przez przeglądarkę złośliwych zasobów. Permissions-Policy to nagłówek, który pozwala witrynie kontrolować, które funkcje i interfejsy API mogą być używane w przeglądarce. On również nie jest użytkowany. Są to przykłady tego, jak odpowiednie nagłówki ustawione na odpowiednie wartości mogą wzmocnić bezpieczeństwo aplikacji webowej. Ich prawidłowa konfiguracja i użytkowanie jest jednym z podstawowych i najczęściej używanych sposobów na zwiększenie bezpieczeństwa. W internecie istnieją serwisy, dzięki którym można sprawdzić poziom bezpieczeństwa aplikacji pod względem używanych nagłówków, np. [securityheaders.com](https://securityheaders.com).

### **A07:2021 – Identification and Authentication Failures**

Potwierdzanie tożsamości użytkownika, uwierzytelnianie i zarządzanie sesją jest krytyczne dla ochrony przed atakami związanymi z uwierzytelnianiem. Słabości uwierzytelniania mogą występować, jeśli aplikacja zezwala na zautomatyzowane ataki, takie jak credential stuffing, gdzie atakujący posiada listę ważnych nazw użytkowników i haseł, czy zezwala na ataki typu brute force lub inne ataki zautomatyzowane, dopuszcza stosowanie domyślnych, słabych lub dobrze znanych haseł, takich jak "password123" lub "admin". Jest to jedna z najpopularniejszych kategorii podatności.

W przypadku omawianej aplikacji występują wszystkie wyżej wymienione podatności. Aplikacja zezwala na nielimitowaną ilość prób logowania, co pozwala skutecznie atakować np. za pomocą brute force używając wordlist. W przypadku dużych, komercyjnych aplikacji może to prowadzić do utraty dostępu do kont, kradzież poufnych danych i środków finansowych. Szczególnie niebezpieczne jest to wtedy, kiedy konto administratora nie jest zabezpieczone w dodatkowy sposób. Atakujący może przeprowadzić udany atak brute force na konto administratora, co może spowodować krytyczne straty.

### **A09:2021 – Security Logging and Monitoring Failures**

Kategoria ta ma pomóc w wykrywaniu, eskalacji i reagowaniu na aktywne naruszenia. Bez logowania i monitorowania, naruszenia nie mogą być wykryte. Niewystarczające logowanie, wykrywanie, monitorowanie i aktywne reagowanie to procesy niezbędne do przeprowadzenia przez administratora strony. W najlepszej sytuacji są to procesy ciągłe, bez przerw pozwalających na przeoczenie ewentualnego incydentu. Należy gromadzić i analizować takie sytuacje jak nieudane logowania i adresy, z których były przeprowadzone. Logi najlepiej przechowywać w więcej niż jednym miejscu celem zabezpieczenia przed ich utratą. Szczególną uwagę należy darzyć przeprowadzanie transakcji na wysokie kwoty celem uniknięcia ewentualnych strat.

Omawiana aplikacja nie ma zaimplementowanego żadnego systemu monitorowania bezpieczeństwa. Nie gromadzi nawet logów, informacji o nieudanych logowaniach lub jakiegokolwiek podejranej aktywności. Aplikacja nie może wykrywać, eskalować lub alarmować o aktywnych atakach w czasie rzeczywistym lub zbliżonym do rzeczywistego. Nie posiada także systemu monitorującego walidację danych wejściowych. W przypadku aplikacji biznesowych oraz innych dużych, komercyjnych aplikacji taka sytuacja nie ma prawa mieć miejsca. Atakujący może wykorzystać fakt, że aplikacja nie monitoruje podejranej aktywności i wykorzystać to do przeprowadzenia ataku w sposób całkowicie niepodjęty.

## 6. Podsumowanie

Omawiana aplikacja posiada przynajmniej 5 luk bezpieczeństwa spośród listy OWASP Top Ten. Została napisana właśnie pod tym kątem. Projekt ten pozwolił rozwinąć posiadane umiejętności pisania kodu serwerowego w JavaScript wykorzystując środowisko Node.js oraz utrwalić te posiadane. Szczególnie wartościowe było nauczenie się zwracania uwagi na bezpieczny kod już podczas pisania, a nie tylko podczas przeglądu w pełni napisanej aplikacji. Odświeżone zostało kilka umiejętności, takich jak chociażby korzystanie z programu Postman oraz pracy z bazą danych MongoDB.

Wartościowe było zapoznanie się z lukami bezpieczeństwa OWASP Top Ten na stronie [owasp.org](https://owasp.org), co pozwoliło na poznanie nowych rodzajów podatności oraz zapoznanie się ze statystykami występowania takich luk powszechnie. Z racji tego, że w projekcie została użyta lista OWASP Top Ten w najnowszym wydaniu z 2021 roku, pozyskano najświeższe informacje o kategoriach podatności.

Projekt ten był bardzo dobrym punktem wyjścia do pracy jako specjalista do spraw cyberbezpieczeństwa. Aplikacje sieciowe z roku na rok są coraz częściej używane. Wzrost ich popularności oraz ich rozwój stawiają nowe wyzwania w obszarze bezpieczeństwa informatycznego, co wymaga od aktualnych specjalistów ciągłego rozwoju w tej dziedzinie, a od studentów wymaga się nauki najnowszych technologii, procedur, sposobów pisania kodu, rodzajów i metod audytu bezpieczeństwa tak, aby posiadana wiedza i umiejętności były aktualne podczas wejścia na rynek pracy.

Koniec