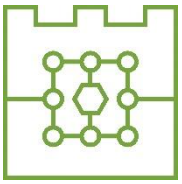
Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji



Patryk Gajewski

Numer albumu: 125715

Analiza wybranych aspektów bezpieczeństwa autorskiego systemu rezerwacji tematów prac dyplomowych

The selected security aspects analysis of the proprietary thesis topic booking system

Praca magisterska
na kierunku INFORMATYKA
specjalność CYBERBEZPIECZEŃSTWO

Praca wykonana pod kierunkiem:
dr Radosław Kycia

Kraków, 2023

Spis treści

Streszczenie

W ramach pracy dyplomowej przeprowadzono analizę wybranych aspektów bezpieczeństwa autorskiego systemu informatycznego dedykowanego rezerwacji tematów prac dyplomowych. Analizowany system jest rozwiązaniem webowych złożonym z dwóch odseparowanych części. W ramach analizy sprawdzono odporność systemu na wybrane zagrożenia ujęte w ramach *OWASP Top 10 Web Application Security Risks 2023*. Przeprowadzenie analizy oparto o metodykę manualnych i automatycznych testów penetracyjnych. Jako główne źródłem wiedzy wykorzystano kompendium *OWASP Web Security Testing Guide*. W ramach analizy przeanalizowano poprawność konfiguracji serwera. Sprawdzono odporność rozwiązania na ataki typu: *CSRF*, *XSS, NoSQL Injection* czy *Brute Force*. Zweryfikowano występowanie podatności w zewnętrznych zależnościach. Analizie poddane zostały mechanizmu autentykacji i autoryzacji. Dostęp do kodu źródłowego umożliwił przetestowanie rozwiązań redukujących i eliminujących wykryte w ramach analizy podatności. Wykorzystane techniki i narzędzia pozwoliły zabezpieczyć rozwiązanie przed testowanymi rodzajami ataków na aplikacja internetowe. W pracy wykorzystano narzędzia takiej jak: *Postman*, *Zaproxy*, *Hydra* czy *npm audit*.

Wstęp

1. Cel pracy

Głównym celem pracy było zbadanie wybranych aspektów bezpieczeństwa autorskiego systemu rezerwacji tematów prac dyplomowych stworzonego kilka lat wcześniej. Analiza miała na celu wykrycie źródeł zagrożeń w domenie badanych aspektów bezpieczeństwa. Aspekt badawczy dotyczył również wypracowania rozwiązań pozwalających zredukować lub wyeliminować wykryte w ramach analizy podatności. Autorowi przyświecał cel pogłębienia wiedzy na temat bezpieczeństwa aplikacji internetowych. Wykonywanie testów penetracyjnych i wdrażanie rozwiązań bezpieczeństwa było domeną szczególnie rozważaną w ramach pracy. Celem było pozyskanie wiedzy praktycznej pozwalającej tworzyć bezpieczniejsze aplikacje w przyszłości. Ideą poboczną była chęć zwrócenia uwagi na wieloaspektowość kwestii bezpieczeństwa aplikacji internetowych.

2. Zakres pracy

Zakres pracy obejmował przeprowadzenie serii automatycznych i manualnych testów penetracyjnych pozwalających wykryć luki występujące w rozwiązaniu. Szczególną uwagę poświęcono wybranym zagrożeniom ujętych w ramach *OWASP Top 10 vulnerabilities 2022*. Corocznie wydawany raport zawiera listę 10 najpopularniejszych zagrożeń dla oprogramowania. Zestawienia tworzone są w różnych kategoriach w odniesieniu m. in. do aplikacji mobilnych, aplikacji webowych czy interfejsów *API*. W ramach pracy przetestowano odporność na wybrane zagrożenia dotyczące aplikacji internetowych ale również udostępnionego interfejsu *REST API*. *OWASP (ang. Open Web Application Security Project)* będąca autorem corocznego rankingu jest organizacją non-profit prowadzącą szeroko zakrojone działania mające na celu poprawę bezpieczeństwa oprogramowania. W ramach pracy przetestowano aplikację pod kątem:

- zabezpieczeń dostępu do określonych funkcji,
- poprawności procesu rejestracji użytkownika,
- poprawności konfiguracji serwera,
- odporności na ataki typu *XSS (ang. Cross Site Scripting),*
- odporności na ataki typu *CSRF (ang. Cross Site Request Forgery),*
- odporność na ataki typu *NoSQL Injection,*
- odporności na ataki typu *Brute Force,*
- podatności zewnętrznych bibliotek,
- walidacji danych wprowadzanych przez użytkownika.

Obszar pracy obejmował również przetestowanie i wdrożenie konkretnych rozwiązań pozwalających zredukować lub wyeliminować wykryte zagrożenia. Zaprezentowane sugestie rozwiązań znajdują swoje zastosowanie w odniesieniu do podobnych systemów.

3. Metodyka badawcza

Specyficzność procesu testowania aplikacji internetowych jest powodem istnienia wyłącznie wąskiej grupy formalnych metodyk przeprowadzania testów bezpieczeństwa. Szczególnie jeżeli mowa o metodykach dostarczających konkretne narzędzia[1]. Szeroko wykorzystywane są metodyki *OSSTMM (Open Source Security Testing Methodology)* oraz metodyka zaproponowana przez organizację OWASP. To właśnie wybrane elementy z metodyki zaproponowanej przez organizację OWASP zostaną wykorzystane w niniejszej pracy. Za główne źródło merytoryczne należy uznać *Web*

[1] (Marek Antczak, 2013)

Security Testing Guide w wersji 4.2[2]. Jeden ze sztandarowych projektów organizacji OWASP jest kompendium wiedzy na temat testowania aplikacji internetowych. W ramach kompendium wyróżnione zostały m. in. następujące części[3]:

- Część druga mającą charakter wprowadzający i wysokopoziomowy. W jej treści znajdują się m. in. ogólne wymagania testowania aplikacji internetowych. W tej części zawarto również liczne informacje na temat zasad testowania oraz modelowaniu zagrożeń.
- Część trzecia będąca wprowadzeniem do *OWASP Testing Framework*. OWASP Testing Framework jest jednym z podejść do kwestii implementacji procesu testowania w trakcie trwania całego cyklu życia oprogramowania. W podrozdziałach zastosowano podział treści właśnie ze względu na poszczególne etapy cyklu życia oprogramowania.
- Część czwarta najbardziej praktyczna. Wprowadza do *OWASP Web Application Security Testing Methodology* będącej metodyką testowania aplikacji internetowych. W tej części zawarto liczne praktyczne sposoby testowania aplikacji pod kątem występujących zagrożeń.

Z uwagi na obszerność tego źródła liczącego ponad 400 stron wykonane zostaną tylko wybrane z przedstawionych testów w odniesieniu do konkretnych komponentów aplikacji. Najnowsza na aktualną chwilę wersja źródła to wersja 4.2. Została opublikowana w grudniu 2020 roku.

Każdy wykonany test składać się będzie z trzech zasadniczych etapów: przygotowania do testu zawierającego wprowadzenie merytoryczne, przeprowadzenie testu oraz analiza uzyskanego wyniku. Podejście będzie wykorzystane zarówno względem testów manualnych i automatycznych. W dalszej części pracy zostaną przedstawione rozwiązania pozwalające zredukować lub wyeliminować zagrożenie.

[2] (OWASP, 2020)
[3] (Dyjak, 2021)

Część teoretyczna

1. Wprowadzenie do terminologii bezpieczeństwa

Termin bezpieczeństwo wywodzi się od łacińskiego słowa „securitas„. Zwrócić należy uwagę, że słowo „securitas" jest połączenie dwóch wyrazów: „sine" oraz „cura"[4]. „Sine" tłumaczone jest jako „bez" natomiast „cura" tłumaczone jest jako „lęk" lub „obawa". „Securitas" tłumaczone może być więc jako „bez lęku". Pojęcie bezpieczeństwa ma charakter wielowymiarowy. Rozumiane może być odmiennie w zależności od kontekstu a także autora. Bezpieczeństwo w wymiarze ogólnym rozumiane jest jako stan bez troski, stan braku poczucia zagrożenia, stan wolny od niepokoju, stan pewności[5]. Szerokie spektrum znaczeniowe terminu było powodem powstania odseparowanej dziedziny nauk. Nową dziedzinę nauk zajmującą się badaniem tej domeny nazwano *securitologią*.[6] *Securitologia* zaliczana jest do grona nauk praktycznych. Zajmuje się badaniem zagrożeń egzystencji człowieka, jego prawidłowego rozwoju i swobodnego funkcjonowania. Podmiotem badań są również organizacje społeczne.[7] Definicja terminu bezpieczeństwa wraz z postępem rozszerzała swój kontekst znaczeniowy. Problematyka bezpieczeństwa ma tak długą historię jak historia rodzaju ludzkiego[8]. Człowiek od zarania dziejów zmagał się bowiem z różnego rodzaju zagrożeniami, które ewoluowały na przestrzeni czasu. Obok zagrożeń ze strony środowiska człowiek ciągle narażony był na różnego rodzaju wątpliwości wewnętrzne, duchowe a także te natury psychologicznej.

Obecnie występuje mnogość interpretacji terminu. Według R. Zięba w ogólnym znaczeniu bezpieczeństwo jest pewnością istnienia oraz przetrwania. Stanem braku zagrożenia względem posiadania oraz funkcjonowania a także swobodnego rozwoju podmiotu.[9] Bezpieczeństwo według J. Stańczyk to stan charakteryzujący się spokojem, stabilnością i gotowością reagowania na występujące zagrożenia. Bezpieczeństwo ma charakter zmienny, na którego poziom mają istotny wpływ występujące zagrożenia a także strategie obierane w celu neutralizacji i redukcji zagrożeń. Bezpieczeństwo może być rozumiane również jako proces. Tym mianem można określić całokształt wysiłków mających na celu przeciwdziałanie potencjalnym zagrożeniom a także ich skutkom. Podejście do bezpieczeństwa jako procesu wydaje się być bardziej naturalne, odzwierciedlające dynamikę zmian w domenie bezpieczeństwa.[10] J. Stańczyk podkreśla, że procesualnym wymiarem bezpieczeństwa jest odzwierciedleniem jego zmienności w czasie.[11] Zmienność w czasie rzuca nowe światło na postrzeganie bezpieczeństwa i naturalnie kieruje w stronę bezpieczeństwa rozumianego jako proces.

Bezpieczeństwo jako pojęcie wieloaspektowe doczekało się przynajmniej kilku podziałów ze względu na jego typ. J. Stańczyk zasugerował podział ze względu na ujęcia takie jak[12]:

- podmiotowe (np. jednostkowe),
- przedmiotowe (np. wojskowe czy informacyjne),
- przestrzenne (np. lokalne, regionalne czy globalne),
- cele (negatywne lub pozytywne),

[4] (Zięba, 2012, str. 7)
[5] (Remigiusz, 2010)
[6] (Fałdowski, 2018, strony 110-111)
[7] (Korzeniowski, Securitologia. Nauka o bezpieczeństwie człowieka i organizacji społecznych, 2008, str. 53)
[8] (Koziej, Wstęp do teorii i historii bezpieczeństwa, 2015, str. 19)
[9] (Zięba, 2012, str. 8)
[10] (Koziej, Bezpieczeństwo: istota, podstawowe kategorie i historyczna ewolucja, 2011)
[11] (Stańczyk, 2017)
[12] (Jerzy, 1996)

Bezpieczeństwo może być rozumiane również jako potrzeba. Bezpieczeństwo ma charakter podmiotowy co oznacza, że odnosi się do konkretnego podmiotu. Podmiotem tym może być jednostka, grupa społeczna, instytucja ale również przestrzeń. Bezpieczeństwo ujęte w kontekście potrzeby zaliczane jest do grona podstawowych potrzeb człowieka. Podobnie jak potrzeby natury fizjologicznej bezpieczeństwo zaliczane jest do grupy potrzeb niższego rzędu w rozumieniu hierarchii Abrahama Maslowa. Spełnienie potrzeb niższych rzędów sprawia, że powstają odpowiednie warunki do realizowania potrzeb wyższych rzędów takich jak np. potrzeba przynależności czy samorealizacji. Znaczenie tej potrzeby w rozwoju jednostek ale i całych społeczeństw sprawiło, że powstały różnego rodzaju regulacje prawne pozwalające zapewniać szeroko rozumiane bezpieczeństwo. Również w Konstytucji RP zawarte są liczne zapisy dotyczące potrzeby zapewnienia bezpieczeństwa obywatelom RP.[13] W zapisach najważniejszego aktu prawnego znajdują się zapisy dotyczące poszczególnych jednostek jak i całego społeczeństwa.

Termin bezpieczeństwo jest blisko związany z pojęciem zagrożenia. W opinii L. Korzeniowskiego zagrożenie jest przyczyną niepożądanego stanu.[14] W rozumieniu ogólnym zagrożenie jest zdarzeniem, którego wystąpienie może mieć negatywny wpływ na istnienie bądź funkcjonowanie określonego podmiotu. Termin zagrożenie ma charakter podmiotowy co oznacza, że zazwyczaj określony jest względem konkretnej domeny. Przykładem może być zagrożenie stanem klęski żywiołowej. W celu zniwelowania zagrożeń podejmuje się różnego rodzaju działania mające na celu ich redukcję i eliminację. W domenie zagrożeń występuje co najmniej kilka podziałów. Jeden z bardziej oczywistych to podział ze względu na zagrożenia subiektywne oraz obiektywne. Zagrożenia subiektywne są efektem postrzegania konkretnej sytuacji przez podmiot jako złowrogie, mogące mieć negatywne konsekwencje. Zagrożenia obiektywne są definiowane na podstawie rzeczywistych informacji niosących mierzalną wartość logiczną. Kolejny podział dzieli zagrożenia ze względu na obszar pochodzenia. Zagrożenia wewnętrzne mają swoje źródło w podmiocie natomiast zagrożenia zewnętrzne pochodzą spoza danego podmiotu. To właśnie ten podział stosowany jest m. in. w odniesieniu do bezpieczeństwa państwowego.

W kontekście bezpieczeństwa nie sposób nie wspomnieć o ryzyku, które definiowane jest jako funkcja zagrożenia i prawdopodobieństwa jego wystąpienia[15]. Wyeliminowanie większości zagrożeń nie jest w pełni możliwe natomiast zmniejszenie prawdopodobieństwa jest wymierną metodą pozwalającą minimalizować konkretne ryzyko.

2. Wprowadzenie podstawowej terminologii domeny IT

API (z ang. Interfejs programowania aplikacji) jest to zbiór zasad regulujących sposób komunikowania się ze sobą programów lub ich fragmentów[16]. Jest rodzajem interfejsu oprogramowania oferującym wykonywanie usług innym modułom oprogramowania.

REST (ang. Representational State Transfer) jest stylem architektury oprogramowania opierającym się o zbiór określonych reguł opisujących sposoby definicji zasobów oraz dostępu do zasobów.

REST API jest interfejsem oprogramowania zbudowanym w oparciu o zasady *REST*. Interfejs *API* może być określony jako *RESTful* w sytuacji kiedy spełnionych jest łącznie kilka warunków.

[13] (Konstytucja Rzeczypospolitej Polskiej)
[14] (Korzeniowski, Securitologia. Nauka o bezpieczeństwie człowieka i organizacji społecznych, 2008, str. 58)
[15] (Korzeniowski, Menedżement, 2010)
[16] (Wprowadzenie do REST API, 2023)

Według D. Szczepaniak tymi warunkami są odpowiednio[17]:

- separacja warstwy frontendowej od części backendowej
- bezstanowość
- wykorzystywanie mechanizmu cache
- jednoznaczne wskazywanie zasobów przez strukturę adresów
- warstwy dostępu do danych, logiki biznesowej i prezentacji są odseparowane

Frontend w odniesieniu do aplikacji internetowych część systemu udostępniająca widoczny dla użytkownika interfejs graficzny. Rolą tej części systemu jest pobieranie danych wprowadzanych przez użytkownika w postaci różnego rodzaju formularzy. Pobrane w ten sposób dane są następnie grupowane w zapytania, które za pośrednictwem odpowiednich protokołów przesyłane są do części backendowej.

Backend w odniesieniu do aplikacji internetowych jest częścią odpowiedzialną za przetwarzanie odebranych zapytań (żądań). Przetworzenie żądań najczęściej sprowadza się do wykonania określonych operacji na danych przechowywanych w bazie danych i zwróceniu wyniku. W ramach backendu często udostępniany jest interfejs, który pozwala na komunikację między częścią backendową a frontendową.

Token jest przedmiotem będącym w własnością jednej jednostki pozwalający na jej jednoznaczną identyfikację.[18]

JSON Web Tokens ogólnodostępny standard pozwalający na wymianę informacji pomiędzy dwiema frakcjami w postaci zaszyfrowanej. Informacje przekazywane są w postaci obiektów *JSON* Web Signature lub w postaci struktur *JSON Web Encryption*. *Token JWT* składa się z nagłówka, zawartości oraz sygnatury. W ramach tokenu przekazywane są informacje podpisane cyfrowo w sposób jawny bądź niejawny. Do jawnego podpisania wykorzystywane są klucze *RSA* lub *ECDSA*. W przypadku podpisu niejawnego wykorzystywany jest algorytm *HMAC*.[19]

Middleware jest oprogramowaniem pośredniczącym występującym pomiędzy warstwami oprogramowania. W odniesieniu do frameworka *Express* są to funkcje uruchamiane w momencie odebrania żądania *HTTP* skierowanego do serwera. Funkcja będąca *middleware* ma dostęp do danych zawartych w *HTTP* request oraz response. Dzięki temu funkcja ma możliwość zakończyć żądanie HTTP w sytuacji gdy jego dalsze przetwarzanie nie ma sensu lub nie jest możliwe. Funkcja może również przekazać otrzymane żądanie HTTP do następnej funkcji *middleware*. Funkcje tego typu wykorzystywane są szczególnie często w przypadku walidacji, sanityzacji, autoryzacji czy autentykacji.

3. Wprowadzenie do terminologii cyberbezpieczeństwa

Cyberprzestrzeń jest definiowana przez amerykański Departament Obrony jako globalna domena środowiska informacyjnego, na którą składają się współzależne sieci tworzone przez infrastrukturę technologii informacyjnej. Częścią tej domeny są sieci telekomunikacyjne, sieć Internet a także korzystające z nich systemy komputerowe.[20]

Cyberprzestępczość należy do najbardziej rozpoznawalnych terminów pozwalających opisać nowoczesne formy przestępczości komputerowej.[21] Należy zwrócić uwagę na fakt iż pojęcie nie należy do zbioru wyrażeń ustawowych w stanie prawnym obowiązującym na moment tworzenia

[17] (Wprowadzenie do REST API, 2023)
[18] (Narodowy Standard Cyberbezpieczeństwa, 2021, str. 245)
[19] (Siczek, 2019)
[20] (1-02, 2010, str. 58)
[21] (Wasilewski, 2017)

niniejszej pracy.[22] Definicja tego pojęcie została opracowana m. in. przez X Kongres ONZ. W ramach tego wydarzenia przyjęto, że cyberprzestępstwo może zostać ujęte w wąskim i szerokim znaczeniu. Odpowiednio w ujęciu wąskim oznacza wszystkie bezprawne działania, przeprowadzane w ramach operacji elektronicznych. Działania te mają negatywnie wpłynąć na bezpieczeństwo systemów lub procesów wykonywanych przez systemy danych. W ujęciu szerokim obejmuje wszystkie bezprawne działania, które zostały popełnione z wykorzystaniem lub dotyczą systemów lub sieci komputerowych.[23] W polskim prawie występuje mnogość aktów prawnych opisujących przestępstwa zaliczane do grona cyberprzestępstw. Są to m. in:.

- Ustawa z dnia 6 czerwca 1997 r. Kodeks karny (Dz. U. 1997 nr 88 poz. 553).
- Ustawa z dnia 16 lipca 2004 r. Prawo telekomunikacyjne (Dz. U. 2004 nr 171 poz. 1800).
- Ustawa z dnia 5 sierpnia 2010 r. o ochronie informacji niejawnych (Dz. U. 2010 nr 182 poz. 1228).

Nie sposób przeprowadzić studium zorientowanego na wykrywaniu podatności odpowiedniego wprowadzenia. *Podatność* w kontekście pracy rozumiana będzie jako słabość, która może zostać wykorzystana przez zagrożenie. Podatność cechuje kilka determinant:

- podatność ma swoją przyczynę (takim źródłem w kontekście systemów informatycznych może być wykorzystanie zewnętrznej biblioteki posiadającej określone luki),
- podatność posiada określoną skalę (podatności mogą mieć krytyczne znaczenie lub nie wpływać na bezpieczeństwo rozwiązania),
- podatność jest zmienna w czasie (podatność może występować tylko w określonym czasie np. kiedy maszyny serwerowe zostały przeciążone ilością operacji do wykonania),
- podatność jest zależna od siły, którym dysponuje zagrożenie (im atakujący posiada większą siłę tym nawet mniejsze podatności mogą okazać się tragiczne w skutkach).

Cyberatak zwany również atakiem cybernetycznym jest to działanie, którego celem jest materializacja zagrożenia w domenie cyberprzestrzeni. Atak ma na celu wyrządzenie konkretnej szkody i jest działaniem celowym. Pomyślne przeprowadzenie ataku może mieć katastrofalne skutki i prowadzić może nawet do bankructwa atakowanego podmiotu.

Kolejnym z pojęć wykorzystywanym na łamach pracy jest cykl życia oprogramowania. Wyróżnia się kilka modeli cyklu życia oprogramowania. Jednym z nich jest model kaskadowy. Nazwa tego modelu pochodzi od wyrazu waterfall (z ang. wodospad). W ramach modelu wyróżniono następujące fazy cyklu życia oprogramowania[24]:

1. Analiza wymagań,
2. Projektowanie,
3. Implementacja,
4. Testowanie,
5. Wdrożenie,
6. Utrzymanie i wsparcie.

Modelowanie zagrożeń jest techniką polegającą na identyfikacji potencjalnych zagrożeń, na które narażone będzie rozwiązanie. W jej zakres wchodzą również rekomendacje pozwalające na

[22] (Siwicki, 2013)
[23] (Wróbel, 2014, str. 75)
[24] (Nader, 2023)

zmniejszenie lub eliminację wpływu ryzyk na przyszły produkt.[25] Poprawne wykorzystanie techniki pozwala osiągać cele związane z bezpieczeństwem na wczesnych etapach cyklu życia projektów. W temat modelowania zagrożeń zaangażowane są ogromne podmioty czego przykładem jest cykl projektowania zagrożeń *The Security Development Lifecycle* opracowany przez firmę Microsoft. Cykl bazuje na modelu *STRIDE*. Nazwa modelu jest akronimem złożonym z nazw głównych typów zagrożeń[26]:

1. ***S**poofing* (z ang. podszywanie się),
2. ***T**ampering* (z ang. manipulacja),
3. ***R**epudiation* (z ang. wyparcie się),
4. ***I**nformation disclosure* (z ang. ujawnienie informacji),
5. ***D**enial of Service* (z ang. odmowa dostępu do usługi),
6. ***E**levation of Privilege* (z ang. zwiększenie poziomu uprzywilejowania).

Model stosowany jest w odniesieniu do elementów składowych diagramów *DFD* (z ang. diagram przepływu danych). Weryfikacja poszczególnych elementów pozwala wyczerpać temat analizy potencjalnych zagrożeń względem danego komponentu.[27]

4. Bezpieczeństwo w kontekście systemu informatycznego

Bezpieczeństwo w domenie systemu informatycznego skupia się wokół informacji, która jest przesyłana, przetwarzana i przechowywana. Wszelkie zadania realizowane na rzecz bezpieczeństwa dotyczą zapewnienia bezproblemowego przesyłu informacji, poprawnego jej przetwarzania oraz bezpiecznego przechowywania informacji. *Bezpieczeństwo informacyjne* jest pojęciem opisującym tę domenę. W ujęciu K. Liedela pojmowane jest jako ochrona informacji przed niepożądanym ujawnieniem, modyfikacją, zniszczeniem lub zablokowaniem możliwości jej przetwarzania.[28] Jest to pojęcie ogólne obejmujące wszystkie formy wykorzystywane w celu wymiany informacji między podmiotami. Bardziej szczegółowe jest pojęcie bezpieczeństwa teleinformatycznego, którego celem jest ochrona informacji przetwarzanej, przechowywanej i transportowanej za pośrednictwem systemów teleinformatycznych przez działaniami niepożądanymi. Do tych działań należy ujawnienie informacji, jej modyfikacja czy zniszczenie.[29]

W kontekście bezpieczeństwa informacji wyróżnia się 3 podstawowe atrybuty[30]:

- poufność (dostęp jest ściśle ograniczony),
- integralność (informacja przechowywana jest w niezmienionej postaci),
- dostępność (istnieje możliwość dostępu do informacji).

Z uwagi na podmiotowość bezpieczeństwa w niniejszej pracy termin bezpieczeństwa rozpatrywany będzie w kontekście systemu informatycznego. Systemy informatyczne podobnie jak inne podmioty narażone jest na różnego rodzaju zagrożenia, które mogą negatywnie wpływać na jego bezpieczeństwo. W tym miejscu warto przedstawić przynajmniej jedną definicję opisującą systemy uznawane za bezpieczne. S. Garfinkel definiuje system informatyczny za bezpieczny jeżeli

[25] (Wolski, 2019)
[26] (Kruk, 2021)
[27] (Kruk, 2021)
[28] (Liedel, 2008, str. 19)
[29] (Liderman, 2002)
[30] (Andress, 2021, str. 21)

użytkownik może mu zaufać a jego działanie odzwierciedla specyfikację[31]. Przytoczona definicja jest bardzo ogólna natomiast w swojej prostocie zawiera całą głębię.

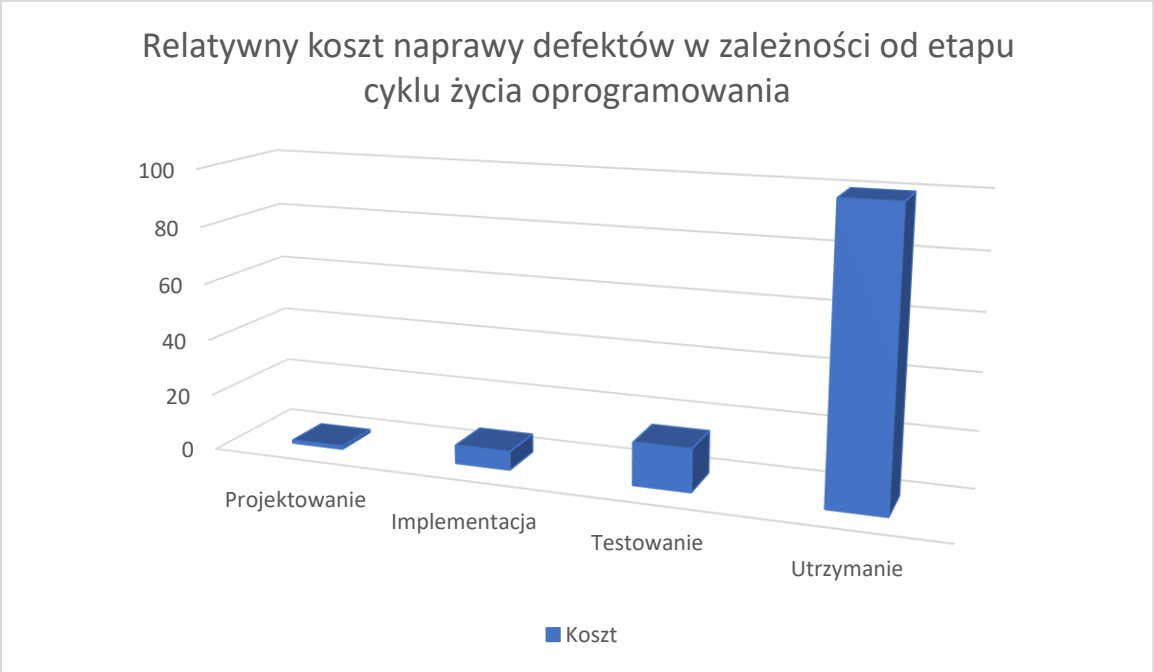
Podobnie jak w ujęciu ogólnym również w przypadku systemów informatycznych zagrożenia możemy podzielić ze względu na pochodzenie oraz celowość. Wyróżnia się następujące pochodzenia zagrożeń w kontekście systemów informatycznych[32]:

1. działania ludzi,
2. awarie urządzeń informatycznych,
3. uchybienia i braki organizacyjne,
4. zdarzenia losowe.

W niniejszej pracy nacisk postawiony został na umyślne działania ludzkie oraz wykrycie uchybień, które mogły zostać popełnione w trakcie różnych etapów tworzenia oprogramowania. W dalszej części pracy celowe i negatywnie nastawione działania nazywane będą atakami.

5. Bezpieczeństwo na poszczególnych etapach życia oprogramowania

Wielokrotnie w ramach *Web Security Testing Guide* autorzy zwracają uwagę na potrzebę obecności kwestii związanych z bezpieczeństwem na każdym z etapów cyklu życia oprogramowania.[33] Pozwala to w sposób zrównoważony podchodzić nie tylko do kwestii bezpieczeństwa ale również do rosnących kosztów usuwania błędów w oprogramowania wraz z kolejnymi etapami cyklu życia oprogramowania. Na rysunku nr 1 zaprezentowano wykres obrazujący jak zmieniają się relatywne koszty usuwania błędów w oprogramowaniu z podziałem na kolejne etapy cyklu życia oprogramowania.



Rysunek 1: Wykres względnego kosztu naprawy usterek w oprogramowaniu na podstawie danych pochodzących z IBM System Science Institute (źródło oryginału: https://www.researchgate.net/figure/IBM-System-Science-Institute-Relative-Cost-of-Fixing-Defects_fig1_255965523)

[31] (Simson Garfinkel, 2003)
[32] (Madej, 2010, strony 83-84)
[33] (OWASP, 2020, str. 14)

Dane do wykresu zostały opublikowane przez firmę IBM[34] – będącą prekursorem pośród przedsiębiorstw domeny IT. Wykres zaprezentowany na rysunku nr 1 pokazuje korelację wzrostu relatywnego kosztu usuwania usterek wraz z kolejnymi etapami życia oprogramowania. Usuwanie błędów okazuje się najbardziej kosztowne na etapie utrzymania produktu i stanowi punkt odniesienia dla pozostałych etapów. Usuwanie błędów na etapie testowania, implementacji, projektowania okazuje się odpowiednio 6, 15 oraz 100 razy tańsze w porównaniu do etapu utrzymania projektu. Pośród wszystkich błędów można założyć, że błędy związane z kwestiami bezpieczeństwa kosztuję firmy najwięcej, ponieważ powodują największy spadek nieocenionego zaufania klienta.

Etap planowania ma charakter wysokopoziomowy. W trakcie trwania tego cyklu życia oprogramowania zbierane są m. in. wymagania klienta. Na podstawie zebranych wymagań kreowane są funkcje i cechy przyszłego produktu. W wyniku przeprowadzonego procesu jest spis wymagań, w którym zawarte będę również wymagania dotyczące bezpieczeństwa.

Etap projektowania pozwala przetworzyć spis wymagań w projekt architektury systemu. Na tym etapie kreowane są również interfejsy użytkownika. W odniesieniu do kwestii bezpieczeństwa już na etapie projektowania warto wykonać analizę zagrożeń pozwalającą zidentyfikować zagrożenia. Proces ten określony jest mianem modelowania zagrożeń. Najczęściej na tym etapie podejmowane są istotne decyzje o wyborze konkretnego języka programowania oraz frameworka.

Implementacja oprogramowania podobnie jak inne etapy jest istotna pod względem bezpieczeństwa przyszłego produktu. W trakcie trwania tego etapu dobierane są zewnętrzne biblioteki, które pozwalają realizować założenia funkcjonalne i niefunkcjonalne produktu. Odpowiedni dobór bibliotek ma istotny wpływ na bezpieczeństwo końcowego produktu. Kluczowym czynnikiem wpływającym na bezpieczeństwo rozwiązania na tym etapie jest jakość tworzonego kodu. W odniesieniu do decyzji podjętych w poprzednim etapie nie wystarczy dobór odpowiednich narzędzi – trzeba jeszcze być w stanie odpowiednio z nich skorzystać. W tym celu wykorzystywanych jest wiele praktyk pozwalających podnosić jakość tworzonego kodu. Jednym z najbardziej znanych jest weryfikacja kodu wykonywana przez pozostałych członków zespołu.

Etap testowania w odróżnieniu od pozostałych jest zorientowany na poszukiwanie różnego rodzaju defektów. Im pełniejszy i lepiej przemyślany zestaw testów różnego rodzaju tym większa ilość błędów zostanie wyłapana na stosunkowo „tanim" etapie. W ramach tego etapu oprócz sztandarowych testów jednostkowych czy integracyjnych wykonać można również pierwsze testy penetracyjne. Ten rodzaj testów pozwala zidentyfikować luki w bezpieczeństwie.

Etap wdrożenia polega na uruchomieniu rozwiązania w środowisku produkcyjnym. Charakteryzuję się zwiększoną ilością pracy osób odpowiedzialnych za uruchomienie i skonfigurowanie środowiska. Poprawna konfiguracja środowiska uruchomieniowego ma kluczowe znaczenie. Zaniedbanie tych kwestii może udaremnić wszystkie wcześniejsze wysiłki. Często w celu zyskania większej pewności etap ten kończy się testami powdrożeniowymi.

Zapewnienie bezpieczeństwa po poziomie utrzymania wymaga podjęcia wielu wysiłków. Szczególnie istotną rolę odgrywa monitoring. Obecne na rynku rozwiązania oferują funkcjonalności takie jak wykrywanie anomalii, kategoryzacja występujących błędów oraz raportowanie zagrożeń. Monitorowanie to tylko jedno z działań na rzecz bezpieczeństwa w fazie utrzymania. Bezpośrednią odpowiedzią na wykryte błędy jest proces wydawania nowych wersji. Dzięki temu oprogramowanie staje się co raz bardziej doskonałe choć zapewne nigdy tego celu w pełni nie osiągnie. Na etapie

[34] (Maurice Dawson, 2010, str. 51)

utrzymania wykonywane są również okresowe kopie zapasowe przechowywanych danych pozwalające uodpornić rozwiązanie na awarię sprzętu i ataki.

Przytoczone działania podejmowane na rzecz bezpieczeństwa w kolejnych etapach cyklu życia oprogramowania obrazują jak wymagającym i wieloaspektowym procesem jest kwestia zapewnienia bezpieczeństwa aplikacji. Wykonywanie działań na różnych etapach cyklu życia oprogramowania pozwala stworzyć kompletną architekturę bezpieczeństwa.

6. Klasyfikacja zagrożeń cyberprzestrzeni

W domenie klasyfikacji zagrożeń cyberprzestrzeni występuje co najmniej kilka podziałów. W ramach jednego z podstawowych wyróżnia się:

- awarie,
- wypadki,
- ataki.

Założyć można, że dwie pierwsze grupy zależą od czynników technicznych mających z natury charakter losowy. Przykładem może być awaria elementów układów elektronicznych, która występuje losowo. Awarie względem wypadków różnią się pochodzeniem. W przypadku wypadków przyczyna pochodzi z zewnątrz. Awarie mają charakter wewnętrzny.[35] Ostatnia grupa dotyczy celowych działań, których wystąpienie jest związane z działalnością człowieka.

Odmienny podział pozwala pogrupować zagrożenia względem charakteru. W ramach tej grupy można wyróżnić zagrożenia mające techniczny i nietechniczny.

Do grupy zagrożeń technicznych zalicza się m. in.:

- złośliwe oprogramowanie przykładem tej grupy są wirusy, robaki czy ransomware,
- ataki na urządzenia sieciowe,
- ataki na systemy wbudowane,
- ataki na urządzenia mobilne,
- ataki na aplikacje mobilne,
- ataki na portfele kryptowalutowe,
- ataki na infrastrukturę.

Jak podaje GOV.PL do zagrożeń nietechnicznych zaliczyć można[36]:

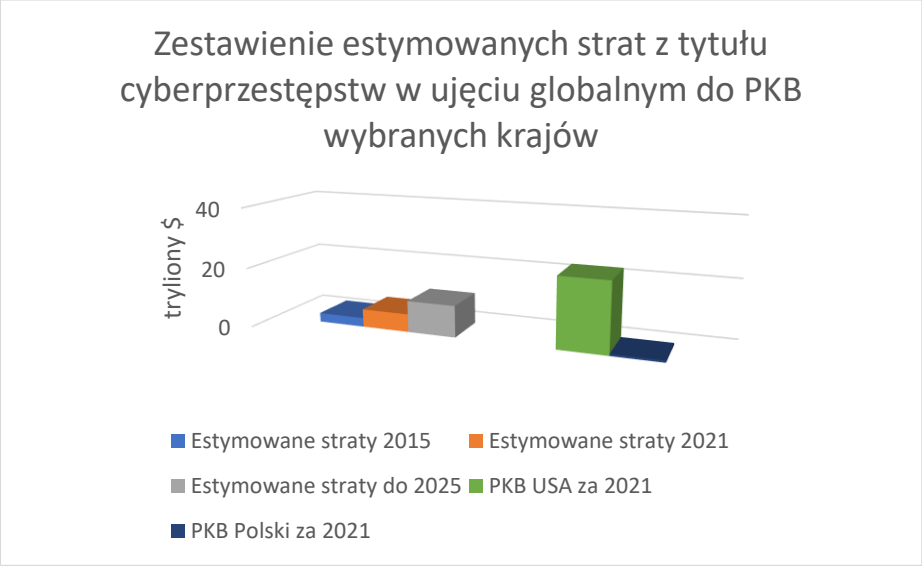
- *Cyberstalking* definiowany jest jako szereg działań najczęściej zorganizowanych, powtarzanych i wskazujących chęć uzyskania kontroli atakującego względem ofiary za pośrednictwem cyberprzestrzeni.
- *Trolling* (z ang. trolowanie) można zdefiniować jako różnego typu nieprzyjazne zachowania nakierowane przede wszystkim w użytkowników forów dyskusyjnych a także mediów społecznościowych. Za motywację tego antyspołecznego zachowania uznawana jest chęć sprowokowania, ośmieszenia lub obrażania innych użytkowników.
- *Flaming* (ang. flame, flame war) rozumiane jest jako prowadzenie wojny na słowa, w której nie obowiązuje jakakolwiek reguła kulturalnej wymiany poglądów

[35] (Uniwersytet Pedagogiczny im. Komisji Edukacji Narodowej, 2019, str. 181)
[36] (Rodzaje cyberzagrożeń - zagrożenia nietechniczne (społeczne), 2023)

- *Cyberprostytucja* będąca zjawiskiem udostępniania materiałów pornograficznych z udziałem udostępniającego w celu uzyskania korzyści majątkowych. Materiałami są najczęściej zdjęcia i filmy ale również pokazy „na żywo".
- *Grooming* czyn przestępczy, którego celem jest zaangażowania dziecka do produkcji materiałów pornograficznych.

7. Skala zagrożeń

Obecnie obserwowany jest co raz szybszy postęp w dziedzinie technologii informacyjnej za jakie można uznać powstanie bardzo zaawansowanych modeli sztucznej inteligencji, rozwój przemysłu 4.0 czy działania polegające na cyfryzacji kolejnych domen codziennego życia. Jak pokazują badania przeprowadzone przez CBOS ciągle rośnie ilość polskich internautów.[37] Tendencja wzrostowa jest jeszcze bardziej widoczna w ujęciu globalnym. Analizując dane zamieszczone na portalu Statista.com grono światowych internautów zwiększyło się o 6% z poziomu 5 bilionów w 2022 do poziomu 5.3 biliona użytkowników w 2023 roku.[38]Do roku 2030 estymuje się, że 7,5 biliona osób będzie korzystało z internetu.[39]Rosnąca liczba użytkowników sprawia, że cyberprzestrzeń staje się co raz bardziej interesującą domeną dla przestępców. Raport stworzony przez CYBERARK na rok 2023 pokazuje, że 89% badanych organizacji było celem przynajmniej jednego ataku w ciągu ostatniego roku. Według raportu sektor ochrony zdrowia najczęściej stanowi cel ataku.[40] Przytoczone dane finansowe określające straty z tytułu cyberprzestępczości pozwolą na lepsze zobrazowanie skali problemu. Według rachunków ujętych w raporcie Esentire straty z tego tytułu za rok 2021 w ujęciu globalnym szacowane są na 6 x 10^{18} $. Przekłada się to na straty poziomu 190.000 $ w każdej sekundzie. Estymuje się, że do 2025 koszt ten zwiększy się do poziomu 10,5 x 10^{18} $.[41] Rysunek nr 2 przedstawia zestawienie szacowanych straty poniesionych w roku 2015, szacowanych straty w roku 2021 oraz szacowanych straty do 2025 roku w porównaniu z *PKB* (produktem krajowym brutto) wybranych krajów za rok 2021.



Rysunek 2 Zestawienie estymowanych strat z tytułu cyberprzestępstw w wybranych latach (opracowanie własne)

[37] (Centrum Badania Opinii Publicznej, 2022)
[38] (Taylor, 2023)
[39] (Morgan, 2022)
[40] (CYBERARK, 2023)
[41] (Morgan, 2022)

Estymowane straty za rok 2021 stanowią ponad 25% PKB USA za rok 2021. Szacuje się, że straty z tego tytułu mogą wzrosnąć o 75% do 2025 roku. Przytoczone dane nie napawają optymizmem i z dużym prawdopodobieństwem ataków będzie przybywać a te mogą okazać się jeszcze bardziej kosztowne.

8. Wybrane rodzaje ataków aplikacji internetowych

Poniżej przedstawiono kilka najbardziej popularnych rodzajów ataków wykorzystywanych w odniesieniu do aplikacji internetowych.

Ataki typu *CSRF* polegają na fałszowaniu żądań pomiędzy witrynami. Jest to rodzaj ataku skierowany w stronę przeglądarki internetowej. W przeprowadzaniu ataku wykorzystywana jest specyfika działania architektury przeglądarki internetowej. Atak ma na celu nakłonienie użytkownika do działania, którego nie zamierzał wykonać. Najczęściej są to akcje dotyczące zmiany adresu email, hasła lub wykonanie przelewu.

W celu zobrazowania specyfiki ataku CSRF cały proces zostanie omówiony na konkretnym przykładzie. Załóżmy, że przykładowy system względem, którego stosowany jest atak typu CSFR składa się z części frontendowej i backendowej. W systemie dostępne są trzy role: gość, użytkownik i administrator. Gość chcąc otrzymać rolę użytkownika musi przejść proces rejestracji, który jest potwierdzany przez administratora. Administrator potwierdza rejestrację użytkownika w specjalnym panelu, w którym ma podgląd szczegółów przesłanych przez użytkownika.

Poszczególne kroki przykładowego ataku *CSRF* mogą wyglądać następująco:

1. Atakujący za pośrednictwem formularza rejestracji jako wartość pola login przekazuje następujący kod:
   **
   Celem ataku jest wykonanie żądania *HTTP* wykorzystując zwiększone uprawnienia użytkownika będącego administratorem.
2. Niczego nieświadomy administrator loguje się do platformy. Następnie przechodząc w panel administratora wyświetla treść pozwalającą zaakceptować nowych użytkowników.
3. W przypadku braku odpowiednich zabezpieczeń przeglądarka będzie próbowała pobrać obrazek. W tym właśnie momencie zostanie wykonane zapytanie opieczętowane sesją użytkownika posiadającego zwiększone uprawnienia.
4. Jeżeli dane zapytanie zostało poprawnie sformułowane to w efekcie zostanie utworzone nowe konto użytkownika z uprawnieniami administratora.
5. W przypadku pomyślnego ataku atakujący wymusił utworzenie konta użytkownika z uprawnieniami administratora.

Ataki typu *XSS (ang. Cross Site Scripting)* polegają na wstrzykiwaniu fragmentów kodu języka skryptowego do przeglądarki ofiary. Wstrzyknięty kod zostanie następnie w niej uruchomiony. Celem ataku jest przede wszystkim kradzież danych pozwalających zidentyfikować tożsamość użytkownika (pliki *cookie*, *tokeny*). Efektem ataku może być przekierowanie na złośliwe strony, modyfikacja danych, podmiana zawartości hiperłączy czy wyświetlanie własnych reklam.[42]

Ataki typu XSS można podzielić na 3 rodzaje[43]:

- *Reflected XSS* – tej rodzaj ataku sprawia, że wygenerowana strona jest odbiciem kodu przekazanego przez atakującego. Poprzez pola formularzy można zamieścić odpowiedni

[42] (Halachev, 2019, str. 163)
[43] (Halachev, 2019, str. 164)

kod *HTML* lub *JavaScript na serwerze*. Użytkownik odczytując złośliwą zawartość uruchamia szkodliwy kod po stronie własnej przeglądarki.

- *Stored XSS* – podobnie i w tym rodzaju ataku złośliwy kod przechowywany jest na serwerze atakowanej aplikacji. Przykładem ataku tego typu może być dodanie komentarza do posta zawierającego złośliwy kod. Każdy z użytkowników odczytujących post będzie ofiarą ataku. W przypadku ataku typu *Stored XSS* obszar rażenia jest zdecydowanie większy w porównaniu do ataku typu *Reflected*.
- *DOM-based XSS* – w odróżnieniu do dwóch poprzednich wstrzykiwanie zachodzi po stronie klienta. Atak polega na odpowiednim spreparowaniu *DOM* (z ang. modelu obiektowego dokumentu).

Ataki typu *Brute Force* polegają na testowaniu poprawności wszystkich możliwych kombinacji znaków w celu odnalezienia poprawnego hasła.[44] Odmianą tego rodzaju ataków są ataki słownikowe. Słowniki są to zbiory kombinacji loginów i hasłem, które będą kolejno testowane. Czas potrzebny na przeprowadzenie skutecznego ataki może się bardzo różnić. Czasochłonność ataku typu Brute Force zależy przede wszystkim od 3 czynników:

- specyfiki hasła – hasło jest kombinacją znaków; Pośród haseł występują kombinacje częściej i rzadziej stosowane. W przypadku kiedy użyto długiego hasła ale należącego do grupy popularnych lub domyślnych haseł ryzyko szybkiego złamania hasła jest duże. Przykładem takiego hasła jest ciąg znaków „1234567890”.
- długości hasła – im hasło jest dłuższe tym jest silniejsze; Narzucenie polityki długiego hasła ma działanie odstraszające. Dłuższe hasła wymagają sprawdzenia większej liczby kombinacji co sprawia, że czas potrzebny do pomyślnego przeprowadzenia ataku znacząco się wydłuża.
- złożoność hasła – hasło może składać się z liczb, małych liter, wielkich liter i znaków specjalnych; Im więcej wykorzystanych grup znaków w haśle tym hasło jest trudniejsze do odgadnięcia.
- zastosowanych metod prewencyjnych.

Aktualnie powszechnie stosowane są rozwiązania uniemożliwiające stosowanie ataków tego typu. Często implementowany jest mechanizm blokady konta po kilku błędnych próbach wprowadzenia hasła. Taka funkcjonalność zaimplementowana została w większości aplikacji bankowych.

SQL Injection polega na wstrzyknięciu do atakowanej aplikacji nieautoryzowanego fragmentu zapytania *SQL*. Taki atak będzie możliwy gdy spełnione są łącznie dwa warunki:

1. Komunikacja silnika bazy danych z danymi opiera się o język SQL lub jego pochodne.
2. Walidacja wprowadzanych danych nie została zaimplementowana lub nie została zaimplementowana w sposób poprawny.

W przypadku, gdy łącznie dwa warunki zostały spełnione fragment zapytania lub podzapytania przekazanego przez atakującego zostanie wykonany po stronie serwera. Taki atak może mieć bardzo poważne konsekwencje począwszy od oczytania zawartości bazy danych skończywszy na wykonaniu kodu w systemie operacyjnym, w którym funkcjonuje baza danych. Przykładem takiego ataku może być sytuacja, w której atakujący wprowadził w pole tekstowe odpowiednio przygotowany fragment zapytania *SQL* a następnie przesłał dane na serwer. Serwer odebrał wprowadzony przez użytkownika tekst jako parametr. Następnie odebrany parametr przekazywany

[44] (Gilberto Najera-Gutierrez, 2019, str. 135)

jest w odpowiednie miejsce w przygotowanym przez programistę zapytaniu. Dzięki temu zamierzone działanie zapytania ulega zmianie i pozwala zwrócić inne informacje. Przed atakami tego typu można uchronić się stosując odpowiednią walidację danych przekazywanych przez użytkownika.

Atak *NoSQL Injection* działa na podobniej zasadzie co atak *SQL Injection*. Celem ataku są rozwiązania wykorzystujących nierelacyjne bazy danych. W związku z wykorzystaniem różnych silników baz danych, języków programowania i frameworków ataki *NoSQL Injection* różnią się od siebie składnią i gramatyką zapytań. W związku z mniej ustrukturyzowaną strukturą ataki na bazy *NoSQL* mogą mieć poważniejsze konsekwencje w porównaniu z atakami *SQL Injection*.

Atak *DoS* (ang. *Denial of Service*) polega na uniemożliwieniu lub spowolnieniu dostępu do zasobów względem uprawnionych podmiotów. Przeprowadzany jest z wykorzystaniem jednego źródła.[45]

Atak DDoS (ang. *Distributed Denial of Service*) cel ataku jest identyczny jak w przypadku *DoS*. Różnica wynika ze sposobu przeprowadzenia ataku. Atak *DDoS* przeprowadzany jest z wykorzystaniem wielu źródeł. Często w tym celu wykorzystywane są komputery zainfekowane specjalnym wirusem tworzące tzw. sieci komputerów *zombi*. Zainfekowane komputery bez pozwolenia użytkownika mogą podejmują określone działania. Przykładem takiej aktywności może być wykonywanie zapytań *HTTP*. Dzięki rozproszeniu źródła odpowiednio przeprowadzany atak jest trudniejszy do wykrycia.

ReDoS jest atakiem odmowy wykonania usługi związanym z wykorzystaniem wyrażeń regularnych. Atakujący wykorzystuje błędną konstrukcję wzorca wyrażenia regularnego. Celem ataku jest doprowadzenie do skrajnych sytuacji, w których serwer wyczerpuje dostępną moc obliczeniową lub pamięć. Wiąże się to ze spowolnieniem lub uniemożliwieniem wykonania usługi.

Man in the middle atak, w którym atakujący wchodzi w rolę pośrednika pomiędzy komunikującymi się podmiotami. W trakcie komunikacji atakujący dokonuje nieuprawnionych modyfikacji przesyłanych danych.[46] Przykładem protokołu narażonego na ataki tego typu jest protokół HTTP przesyłający dane w postaci nieszyfrowanej.

9. Wybrane narzędzia i metody zwiększające bezpieczeństwo systemów

Zapewnienie bezpieczeństwa aplikacji internetowych jest procesem wykorzystującym różnorodne metody i narzędzia. Bez wykorzystania dedykowanych rozwiązań zapewnienie odpowiedniego procesu bezpieczeństwa aplikacji internetowych nie byłoby możliwe.

Do najbardziej znanych narzędzi i metod zakwalifikować można[47]:

- *firewall* (z ang. zapora ogniowa) tworzący barierę w przepływie danych pomiędzy segmentami sieci,
- systemy klasy *IDS* i *IPS,*
- szyfrowanie mające na celu zabezpieczenie danych poprzez wykorzystanie algorytmów kryptograficznych,
- wieloskładnikowa autoryzacja pozwalająca zapobiec przed nieautoryzowanymi operacjami,
- zarządzanie tożsamością i dostępem kontrolujące uprawnienia dostępu użytkowników do zasobów,

[45] (Narodowy Standard Cyberbezpieczeństwa, 2021, str. 90)
[46] (Narodowy Standard Cyberbezpieczeństwa, 2021, str. 154)
[47] (Designveloper, 2022)

- skanery podatności narzędzia będące wykorzystywane w celu automatycznego śledzenia podatności,
- aktualizacje oprogramowania do najnowszych wersji,
- audyty bezpieczeństwa,
- narzędzia pozwalające tworzyć kopie zapasowe danych.

Część praktyczna

1. Opis weryfikowanego systemu

Przyczyn wyboru własnego rozwiązania stworzonego kilka lat wcześniej jako podmiotu analizy było kilka. Jedną z nich był poziom skomplikowania rozwiązania. Złożoność podmiotu analizy pozwoliła przeprowadzić proces analizy z wykorzystaniem różnych technik i narzędzi. Testowanie rozbudowanego rozwiązania pozwoliło bardziej odzwierciedlić proces testowania oprogramowania produkcyjnego. Bardzo istotnym argumentem wyboru własnego rozwiązania był pełen dostęp do kodu źródłowego. Wspomniany dostęp umożliwił przetestowanie rozwiązań mających na celu redukcję bądź eliminację wykrytych zagrożeń. Ostatnim równie ważnym powodem był brak konsekwencji prawnych w odniesieniu do wykonywanie testów penetracyjnych względem podmiotu własnego autorstwa.

System poddany analizie bezpieczeństwa powstał w ramach pracy inżynierskiej zrealizowanej na Politechnice Krakowskiej w 2021 roku pod kierownictwem dr inż. Andrzeja Wilczyńskiego. Głównym założeniem systemu było usprawnienie procesu rezerwacji tematów prac dyplomowych. Stworzone rozwiązanie może zostać z powodzeniem wykorzystywane do zarządzania zbiorem tematów a także biblioteką gotowych prac. System pozwala na wyszukiwanie, tworzenie a także edytowanie własnych propozycji tematów prac dyplomowych a także tematów prac zaproponowanych przez promotorów. Udostępnienie rankingu promotorów jest funkcją wyróżniającą rozwiązanie na rynku. Ranking ten w realny sposób może wpłynąć na świadomość studentów przy podejmowaniu decyzji o wyborze promotora pracy dyplomowej.

System implementuje architekturę *klient-serwer*. Do stworzenia systemu wykorzystano język *TypeScript*, będącym nadzbiorem języka *JavaScript*. Nadzbiór ten oferuje między innymi typowanie statyczne pozwalające na wychwytywanie większej ilości błędów programistycznych na znacznie wcześniejszym etapie. Wykorzystanie *TypeScript* znacząco zwiększyło utrzymywalność kodu. Adnotacje typów czynią kod samodokumentującym się co ułatwia jego zrozumienie i utrzymanie. Na system składają się dwie zasadnicze części: frontendowa i backendowa.

Część frontendową stanowi aplikacja internetowa uruchamiana po stronie przeglądarki klienta po przejściu pod konkretny adres www. Aplikacja udostępnia przyjazny interfejs użytkownika, za pośrednictwem którego użytkownik komunikuje się z serwerem. Wykonywanie określonych zadań w interfejsie powoduje wysyłanie odpowiednich żądań. Żądania przesyłane są za pośrednictwem protokołu *HTTP*. Część frontendowa została wykonana przy pomocy frameworka *React*, który został opracowany w 2011 przez twórców platformy społecznościowej Facebook. *React* od 2013 roku dostępny jest w ramach licencji *open-source*. Zauważyć należy, że rozwiązane jest ciągle szeroko wykorzystywane w projektach takich firm jak Facebook, Instagram, Uber czy Netflix. Przy tworzeniu części frontendowej systemu wykorzystano wiele zewnętrznych bibliotek takich jak m. in. *yup*, *lodash*, *nivo* czy *material-ui*. Interfejs użytkownika oferuje użytkownikowi grupę widoków, do których użytkownik przenoszony jest w ramach routingu zaimplementowanego w aplikacji. Użytkownik po zalogowaniu ma dostęp do widoku listy dostępnych tematów. Widok oferuje również możliwość stworzenia nowej propozycji tematu pracy dyplomowej. Zalogowanemu użytkownikowi udostępniono również widok listy rankingowej promotorów a także widok pozwalający na zarządzanie kontem użytkownika.

Backendowa część systemu odpowiedzialna jest za wykonywanie odpowiednich operacji na przechowywanych danych w zależności od otrzymywanych żądań. Do stworzenia części backendowej wykorzystany został framework *Express*. Rozwiązanie uruchamiane jest w środowisku Node.js. Do składowania danych wykorzystano nierelacyjną bazę danych *MongoDB*. Dane

przechowywane są jako dokumenty pogrupowane w kolekcję, których modelowanie wspomaga biblioteka *Mongoose*. Komunikacja pomiędzy częściami odbywa się za pomocą udostępnionego interfejsu *REST API*. Autentykacja i autoryzacja została oparta o rozwiązanie *JSON Web Token*. W momencie pomyślnego logowania użytkownika zostaje utworzony *JSON Web Token*, który pozwala korzystać z pełnej gamy funkcjonalności oferowanych przez aplikację. Dane zapisane w *JSON Web Token* pozwalają stwierdzić czy użytkownik powinien uzyskać dostęp do zasobu oraz czy posiada odpowiednie uprawnienia do wykonywania określonych akcji na zasobie.

2. Wykorzystywane narzędzia

Standardowym narzędziem wykorzystywanym do zarządzania pakietami w środowisku *Node.js* jest *Node Package Manager (NPM)*[48]. Pozwala w sposób intuicyjny i szybki instalować, aktualizować a także usuwać zależności wykorzystywane w projektach. Zewnętrzne zależności nazywane są bibliotekami lub paczkami. W celu utworzenia nowego projektu, w którym wykorzystywane będą zewnętrzne zależności dostępne w repozytorium *NPM* należy wykonać komendę *npm init*. W wyniku czego powstanie plik *package.json* opisujący cała warstwę zewnętrznych zależności występujących w projekcie. W celu instalowania nowych zależności wykorzystywana jest komenda *npm i dependency-name*. W ramach interfejsu *CLI* (ang. Command Line Interface) dostarczonego przez *NPM* udostępniona została również komenda *npm audit*. Komenda jest narzędziem diagnostycznym pozwalającym na ocenę zależności wykorzystywanych w projekcie pod kątem podatności na ataki. Komenda przesyła zbiór zależności do rejestru a następnie na tej podstawie generowany jest raport znanych luk. W przypadku znalezienia podatności dla każdej z nich obliczony zostanie odpowiedni poziom wpływu na bezpieczeństwo oraz jeżeli są znane podane zostaną środki zaradcze. Podatności pogrupowane zostały pomiędzy 4 poziomy[49]:

- critical – najwyższy poziom sugerujący natychmiastowe zwrócenie uwagi na daną kwestię,
- high – podatności tego poziomu powinny zostać uznane za pilne,
- moderate – podatności tego poziomu mają średnią wagę ,
- low – podatności tego typu mają niską wagę dzięki czemu ich usuwaniem można zająć się w dogodnym czasie.

W kontekście problemu podatność w wersjach zewnętrznych bibliotek należy wyjaśnić zasadę wersjonowania semantycznego. W przypadku formatu ***major.minor.patch*** wersja składa się z 3 składowych[50]:

- **major** – aktualizacja tego poziomu sugeruje, że zaszły w niej zmiany powodujące brak kompatybilności z wersjami niższymi tego poziomu,
- **minor** – aktualizacja tego poziomu oznacza, że dodane zostały nowe funkcjonalności kompatybilne względem wersji poprzednich,
- **patch** – wersja zawiera poprawki, które oferują kompatybilność względem poprzednich wersji.

Zed Attack Proxy (ZAP) jest narzędziem przeznaczonym do testowania bezpieczeństwa aplikacji webowych stworzonym i utrzymywanym przez organizację OWASP. Narzędzie jest darmowe i proste w obsłudze. Część integralną narzędzia stanowi dodatek *Quick Start* umożliwiający rozpoczęcie skanowania aktywnego po wprowadzeniu adresu *URL*. Ten zautomatyzowany sposób pozwala na wykrycie określonych typów zagrożeń. Wykryte zagrożenia zostają pogrupowane w 4

[48] (Wilson, 2013, str. 42)
[49] (NPM, 2016)
[50] (Titus Winters, 2023, str. 464)

stopniowej skali. Każdemu z zagrożeń zostaje przydzielony jeden z poziomów: wysoki, średni, niski oraz informacyjny. W celu wykrycia pozostałych luk konieczne jest przeprowadzenie ręcznych testów penetracyjnych. W wyniku, których można wykryć m. in. luki w kontroli dostępu.

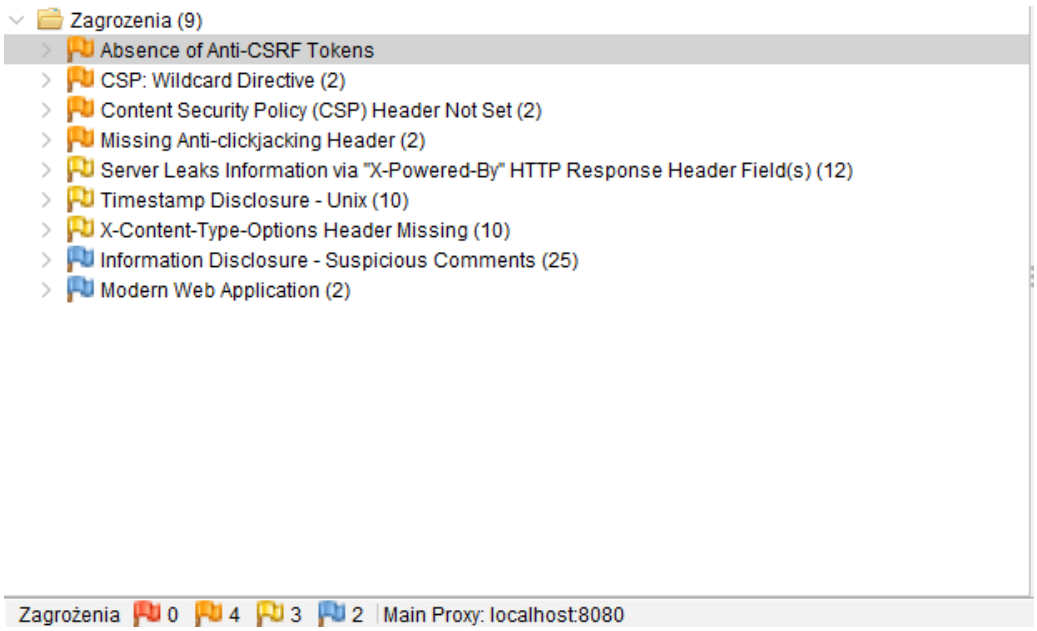
Hydra jest narzędziem konsolowym pozwalającym na przeprowadzenie zautomatyzowanego procesu ataku siłowego celem odgadnięcia kombinacji nazwy użytkownika i hasła. Jest narzędziem chętnie wykorzystywanym zarówno przez pentesterów jak i hackerów.[51] Pozwala łączyć się z usługami sieciowymi takimi jak *FTP*, *SSH*, *Telnet* czy *RDP*. Z powodzeniem może zostać wykorzystany również do ataku na aplikacje internetowe. W kontekście ataku na aplikacje webowe oferuje szereg metod łamania hasła. Pozwala wskazać odpowiednie słowniki zawierające loginy i hasła będące kolejno testowane. Sterując odpowiednimi flagami można zapisywać rezultaty i odpowiednio konfigurować zasady ataku tak aby nie doprowadzić do przeciążenia atakowanego serwera.

Postman to narzędzie wykorzystywane do testowania i dokumentowania *API*. Narzędzie udostępnia przystępny interfejs graficzny, poprzez który można wykonywać różne warianty zapytań. Rozwiązanie pozwala analizować wysłane zapytania jak i odpowiedź serwera. W przyjazny sposób można analizować zawartość zapytania/odpowiedzi, nagłówki czy pliki cookie. Rozwiązanie oferuje tworzenie kolekcji, które pozwala na pogrupowanie zapytań ze względu na źródła.

3. Wykrywanie podatności

1. Analiza narzędziem OWASP ZAP

Na rysunku nr 3 zaprezentowane zostały wyniki automatycznej analizy przeprowadzonej za pomocą narzędzia *OWASP ZAP*.



Rysunek 3 Rezultat automatycznego skanowania aplikacji przez program OWASP ZAP

W ramach procesu skanowania ujawniono łącznie 9 zagrożeń:

- brak tokenów zapobiegających atakom typu *CSRF,*
- brak zastosowania polityki *CSP,*
- brak nagłówka przeciwdziałającego atakom typu *Clickjacking,*
- bdostępnianie informacji na temat serwera (obecność nagłówka *X-Powered-By*),

[51] (Gilberto Najera-Gutierrez, 2019, str. 135)

- brak nagłówka *X-Content-Type-Options,*
- zawartość komentarzy w kodzie, które mogą przechowywać istotne informacje z punktu widzenia atakującego.

Automatyczne skanowanie wyjawiło rzeszę problemów związanych z nagłówkami odpowiedzi serwera wskazując na istotne braki konfiguracyjne. Test nie został zaliczony i będzie wymagał wprowadzenia stosownych zmian w wielu obszarach.

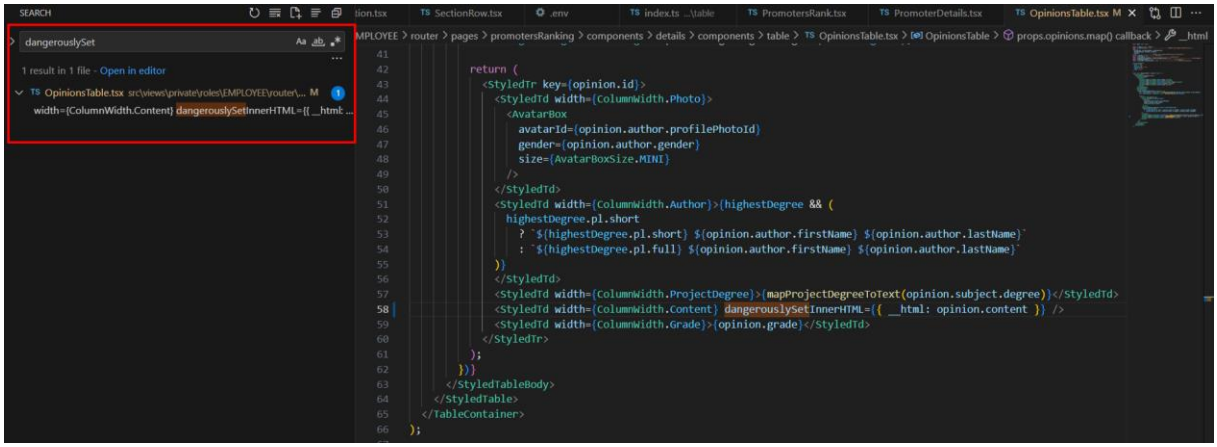
2. Testy odporności na ataki typu XSS

Programiści tworzący rozwiązania we frameworku *React* otrzymali zbiór pewnych technik i narzędzi, których nieodpowiednie wykorzystanie może narażać aplikację na ataki typu *Cross Site Scripting*. Do najważniejszych technik zabezpieczających przed wstrzykiwaniem złośliwego kodu należą:

- sanityzacja danych w przypadku wykorzystywania funkcji *dangerouslySetInnerHTML,*
- uprzednia walidacja adresów *URL* przed wykorzystaniem w aplikacji,
- wykorzystywanie domyślnego powiązania danych poprzez używanie nawiasów klamrowych.

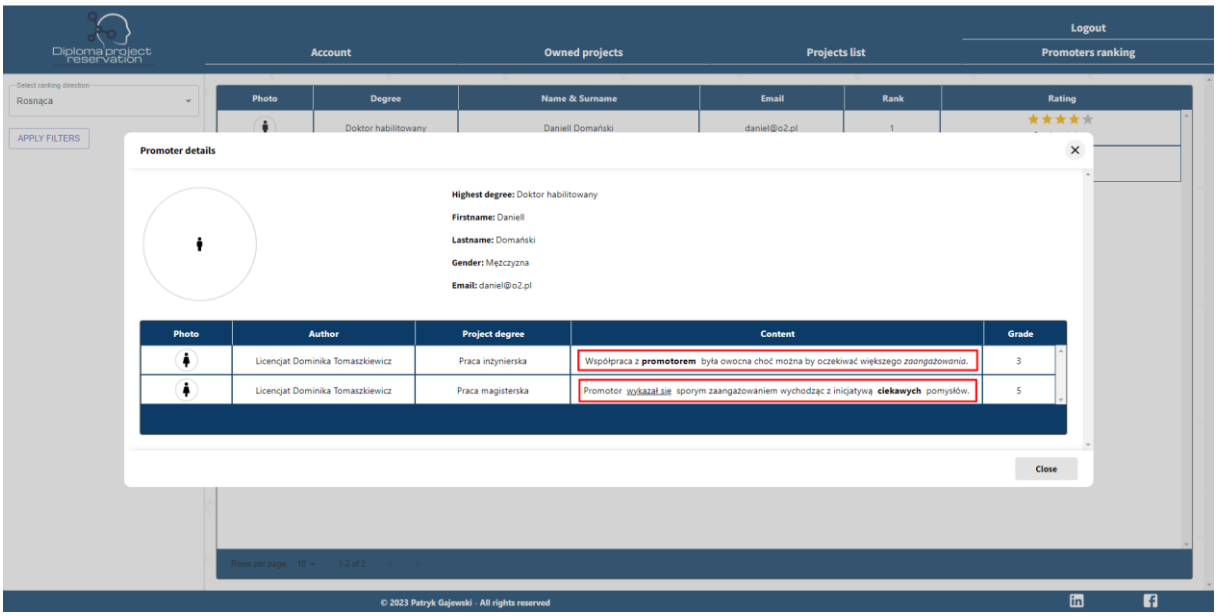
W pierwszej kolejności sprawdzono występowanie funkcji *dangerouslySetInnerHTML*. Sprawdzenie źródeł pozwoliło wykryć pojedyńcze wykorzystania funkcji. Funkcja znalazła zastosowanie w komponencie listy rankingowej promotorów. Umożliwia wyświetlenie sformatowanego tekstu komentarza. Ma to bezpośrednie powiązanie z udostępnieniem tzn. *Rich Text Editor* pozwalającego na formatowanie dodawanego tekstu w ramach komentarza.

Rysunek nr 4 prezentuje fragment kodu aplikacji frontendowej, w której wykorzystana została funkcja *dangerouslySetInnerHTML*.



Rysunek 4 Fragment kodu wykorzystujący funckję dangerouslySetInnerHTML

Rysunek nr 5 prezentuje tabelę zawierająca opinie na temat konkretnego promotora.



Rysunek 5 Element interfejsu użytkownika wykorzystujący funkcję dangerouslySetInnerHTML

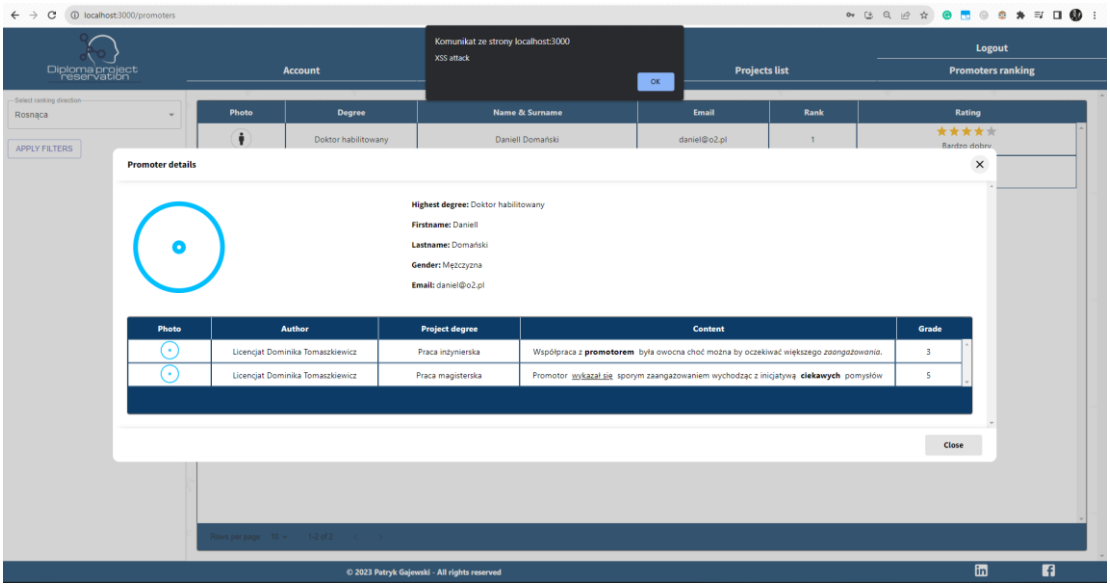
Dla przeprowadzenia testu wprowadzono przykładową złośliwą zawartość do pola komentarza.

```
1  _id: ObjectId('60a8bd012bd9180918702ccc')
2  content: "ych</b>   pomysłów <img src="" onerror="alert('XSS attack');" /> "
3  grade: 5
4  author: 6012ac5c6799561ed1fcdc3d
5  target: 6060e812dec8b45cc682124d
6  subject: 60984e29b96e853523b92f79
7  createdAt: 2021-05-22T08:12:49.441+00:00
8  updatedAt: 2021-05-22T08:12:49.441+00:00
9  __v: 0
```

Rysunek 6 Przykładowa złośliwa zawartość komentarza

Wewnątrz treści komentarz oprócz standardowych znaczników *HTML* takich jak *<b>*, *<u>*, *<i>* został wykorzystany znacznik *<img >*. Podano pusty ciąg znaków jako wartość atrybutu *src*. Dodatkowo do atrybutu *onerror* przekazano złośliwy fragment kodu JavaScript. Podany fragment jeżeli zostanie uruchomiony wywoła okno informacyjne z treścią „XSS attack”.

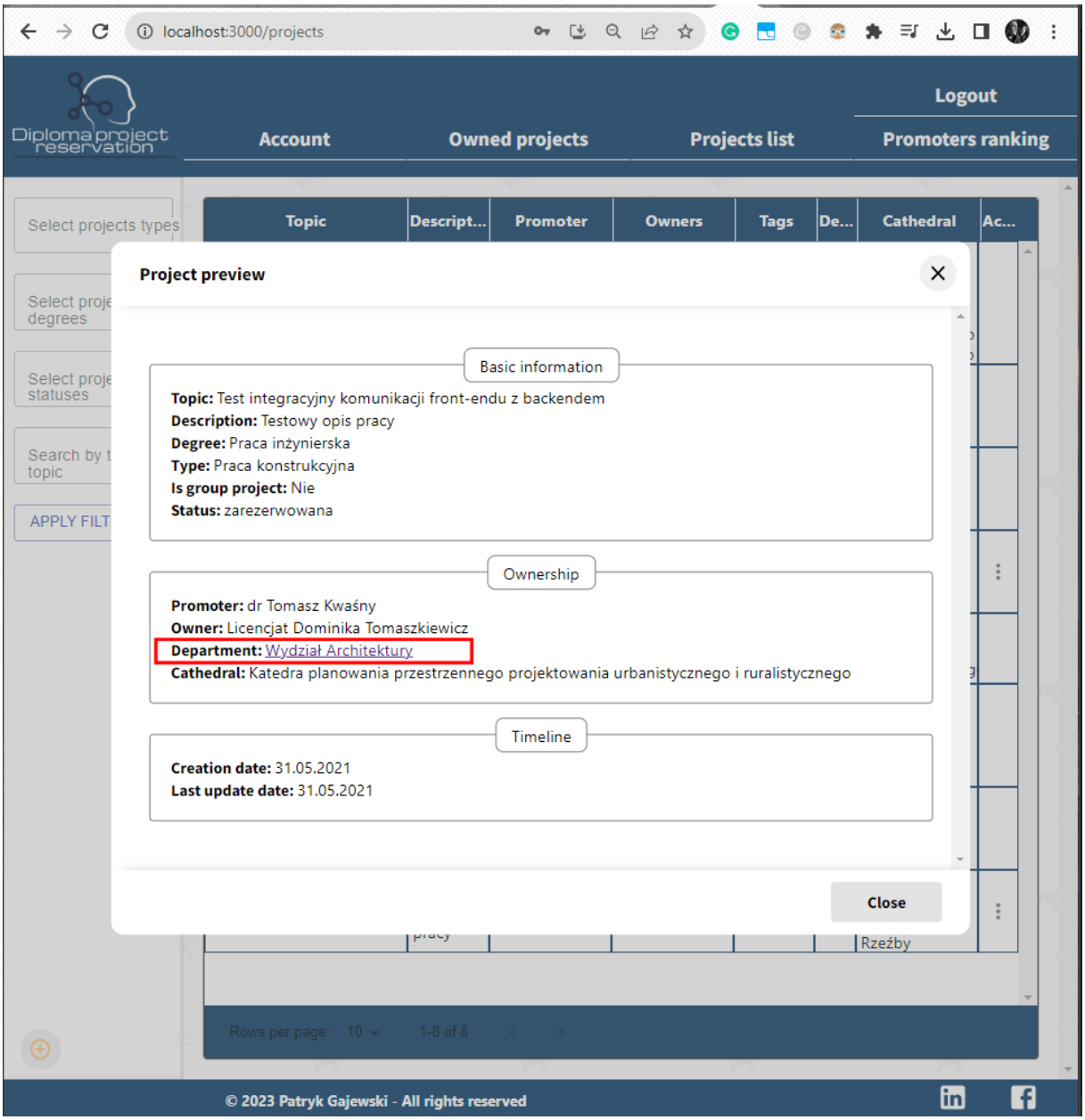
Rysunek nr 7 przedstawia rezultat uruchomienia aplikacji po wprowadzeniu zmian na poziomie bazy danych.



Rysunek 7 Efekt otwarcia widoku detali opinii promotora

Efekt uzyskany na rysunku nr 7 pozwala stwierdzić, że atak został zakończony sukcesem. Po stronie użytkownika jedyną wymaganą akcją było otworzenie określonego widoku. Potencjalna szkodliwość wykrytej podatności jest bardzo duża. Podatność tego typu klasyfikowana jest jako *Stored XSS* i będzie miała wpływa na wszystkich użytkowników wyświetlających zainfekowany widok.

Następnie sprawdzono w jaki sposób przekazywane są adresy *URL* do elementów takich jak *<a>* oraz *<iframe>.* Znaleziona została jedna nieprawidłowość w widoku prezentującym szczegóły tematu pracy.



Rysunek 8 Umiejscowienie wykorzystania otrzymanego hiperłącza

Zauważony błąd dotyczy braku oczyszczenia wartości hiperłącza przed jego wstawieniem na stronę. Rysunek nr 9 przedstawia fragment kodu odpowiedzialny za wyświetlenie hiperłącza.

```
<SectionRow header="Department" content={<a href={props.project.department.website}>{props.project.department.namePL.full}</a>} />
<SectionRow header="Cathedral" content={props.project.cathedral.namePL} />
```

Rysunek 9 Bezpośrednie użycie wartości hiperłącza otrzymanego z backendu

W celu upozorowania ataku odpowiednio zmodyfikowano zawartość hiperłącza w obiekcie reprezentującym wydział.

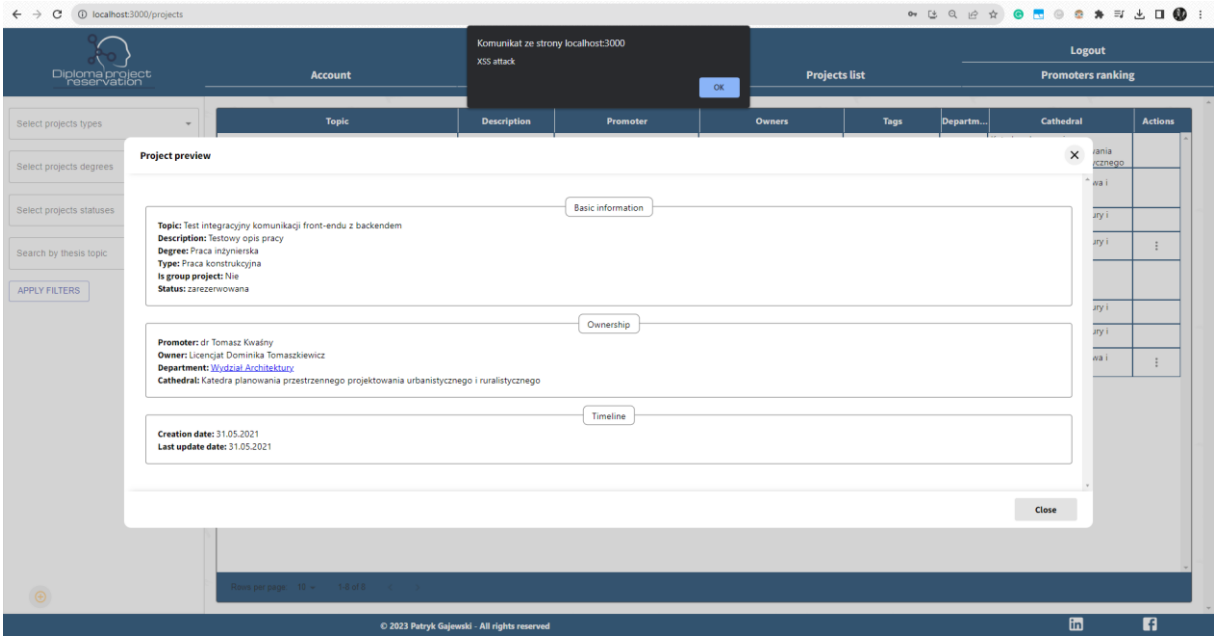
```
_id: ObjectId('6014a0ecd2b0f8eb3408a06d')
▸ cathedrals: Array
▾ namePL: Object
    full: "Wydział Architektury"
    short: "WA"
▸ nameEN: Object
▸ headquarter: Object
▾ links: Object
    website: "http://arch.pk.edu.pl/"
  createdAt: 2021-01-29T23:57:32.400+00:00
  updatedAt: 2021-01-29T23:57:32.400+00:00
```


Rysunek 10 Obiekt reprezentujący wydział przed wprowadzeniem zmiany


```
_id: ObjectId('6014a0ecd2b0f8eb3408a06d')
▸ cathedrals: Array
▾ namePL: Object
    full: "Wydział Architektury"
    short: "WA"
▸ nameEN: Object
▸ headquarter: Object
▾ links: Object
    website: "javascript:alert('XSS attack')"
  createdAt: 2021-01-29T23:57:32.400+00:00
  updatedAt: 2021-01-29T23:57:32.400+00:00
```


Rysunek 11 Zmodyfikowany obiekt reprezentujący wydział zawierający szkodliwy kod

Po wprowadzeniu zmian zaprezentowanych na rysunku nr 11 każdy użytkownik, który wykona akcję kliknięcia klawiszu myszy na hiperłączu uruchomi złośliwy fragment kodu. Ponownie zostało wykryte potencja miejsce wrażliwe na atak *XSS* typu *Stored*.



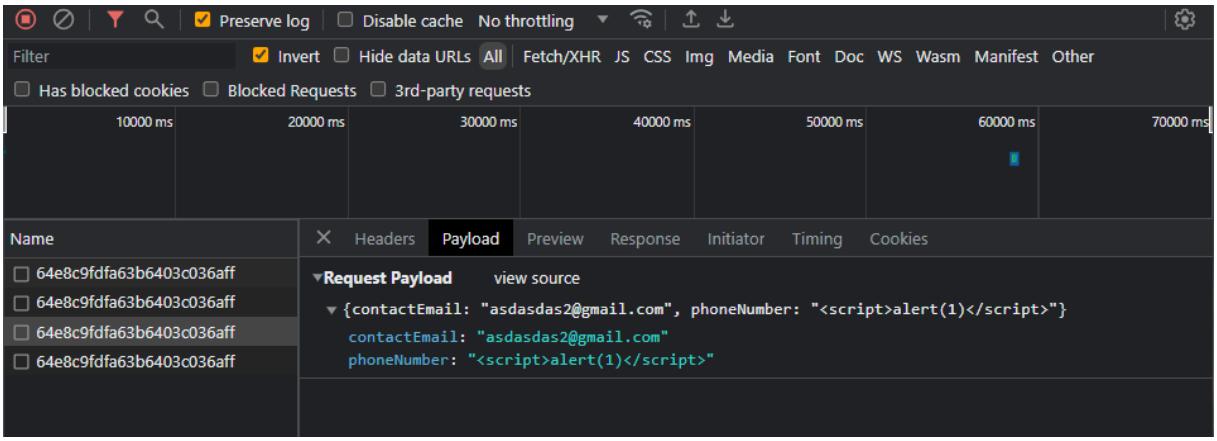
Rysunek 12 Efekt kliknięcia użytkownika w hiperłącze

Przeprowadzony test zakończył się niepowodzeniem. Wymagane będzie podjęcie odpowiedniej akcji.

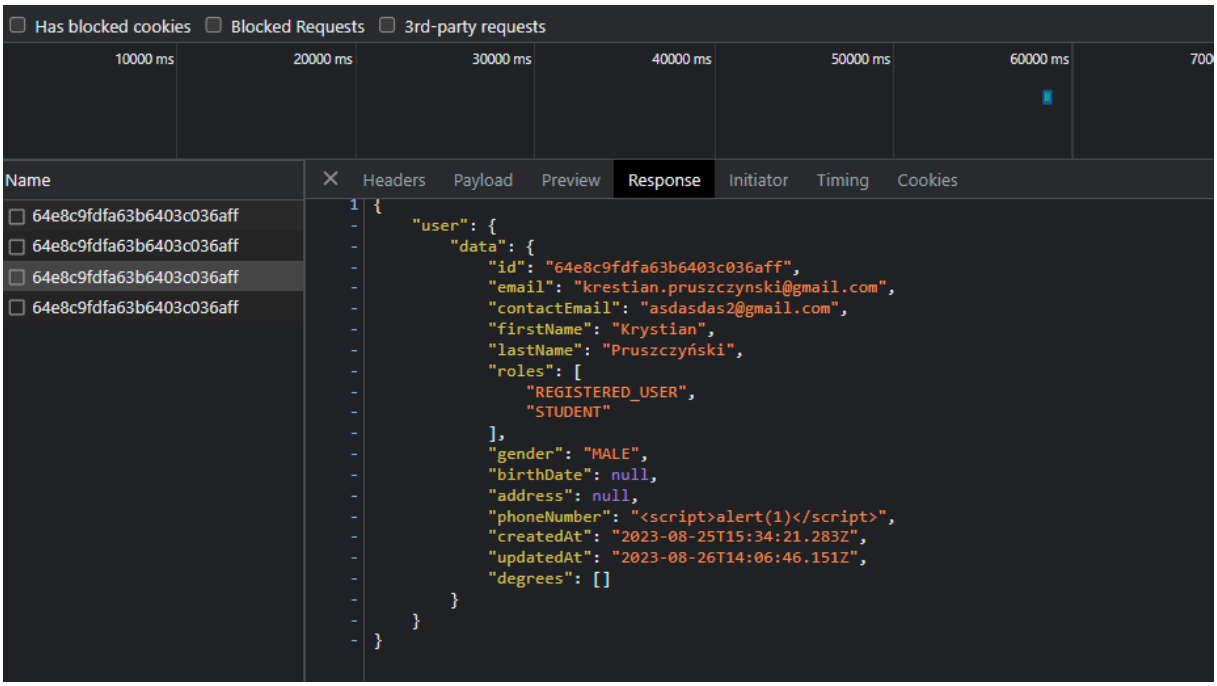
W kolejnym przeprowadzonym teście sprawdzono czy aplikacja odporna jest na przykładowy atak *XSS* typu *Reflected*.

Etapy planowanego ataku:

1. Wprowadzono kod JavaScript do pola przechowującego numer telefonu użytkownika. Kod zawiera instrukcję wywołującą otwarcie okna z komunikatem.
2. Następnie przesłano formularz na serwer.
3. Po aktualizacji danych sprawdzono czy okno z komunikatem ukaże się w przeglądarce



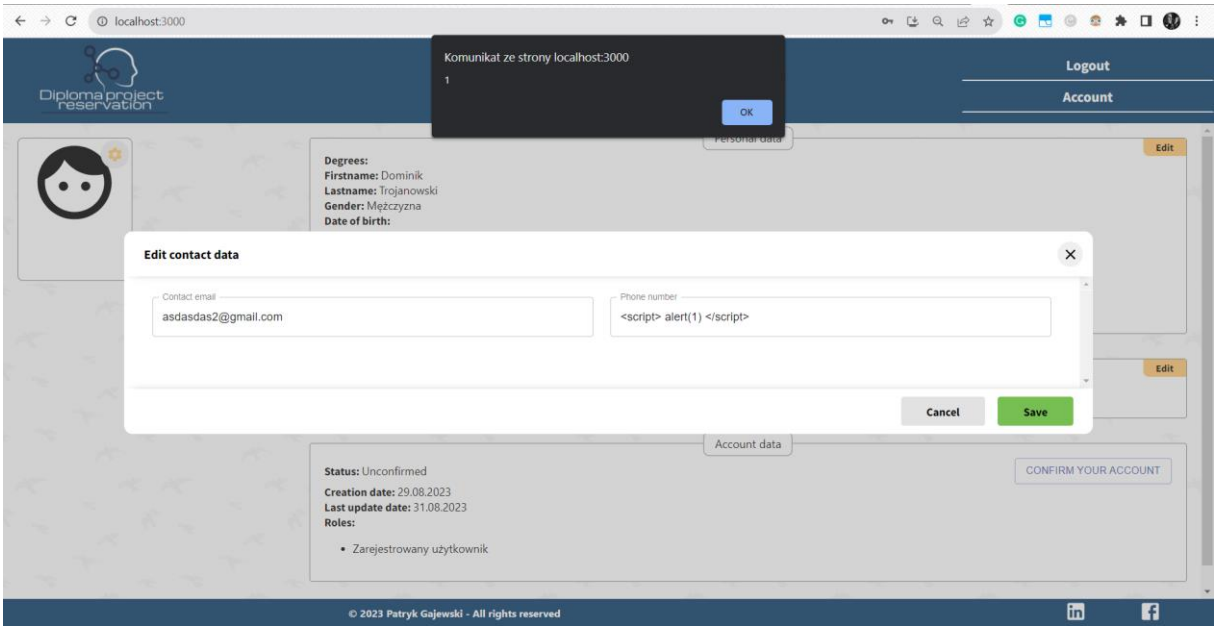
Rysunek 13 Treść zapytania aktualizującego dane kontaktowe użytkownika



Rysunek 14 Odpowiedź serwera na zapytanie aktualizujące dane kontaktowe użytkownika

Na rysunku nr 13 przedstawiono dane przesłane do backendu w ciele zapytania. Jest to podstawą do stwierdzenia braku walidacji danych po stronie przeglądarki. Rysunek nr 14 prezentuje odpowiedź serwera. Odpowiedź sugeruje brak wykorzystania mechanizmu sanityzacji danych. Tajemnicza nazwa określa proces oczyszczania danych z elementów potencjalnie złośliwych elementów bez utraty pierwotnej treści.

Efektem braków walidacji i sanityzacji danych było pokazanie się okna z komunikatem po przejściu na podstronę zarządzania kontem. Rezultat został zaprezentowany na rysunku numer 15.



Rysunek 15 Efekt otwarcia formularza danych kontaktowych po wprowadzeniu szkodliwego fragmentu kodu

Podobne testy przeprowadzono w odniesieniu do pozostałych pól. Te nie przyniosły jednak podobnych rezultatów. Zatem odpowiednia zmiana wymagana będzie jedynie w odniesieniu do pola zawierającego numer telefonu.

3. Test poprawności procesu autoryzacji

W ramach testu sprawdzono czy użytkownik posiadający podstawowe prawa będzie mógł wykonywać akcje zarezerwowane dla użytkowników z większymi uprawnieniami.

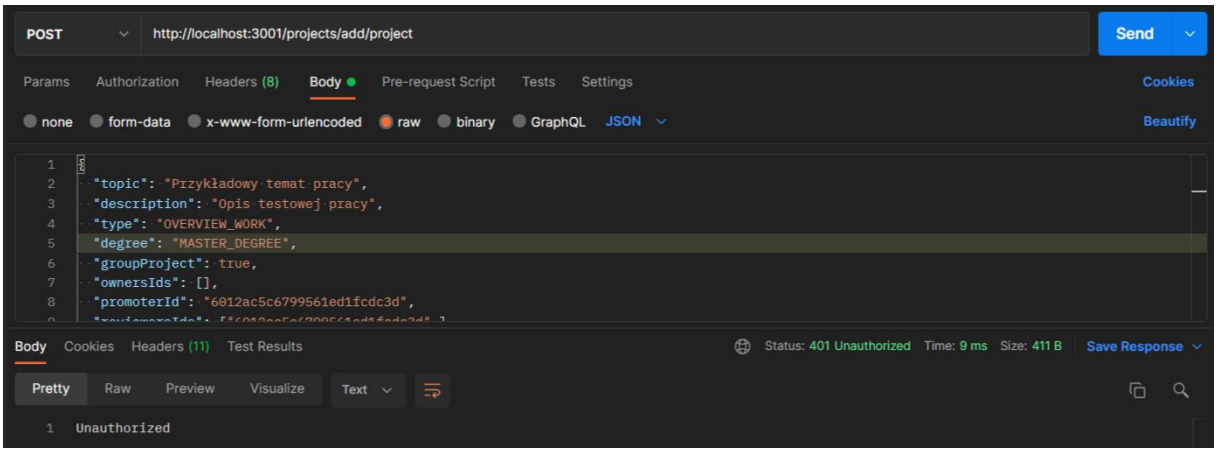
Test składał się z następujących etapów:

1. Utworzenia nowego konta użytkownika, któremu przyznano rolę *REGISTERED_USER*.
2. Przeprowadzenie procesu logowania w wyniku czego został pozyskany *token* pozwalający na identyfikację użytkownika.
3. Wykonania zapytania wymagającego roli *STUDENT* z *tokenem* użytkownika o ograniczonych prawach w celu zweryfikowania czy proces zostanie przeprowadzony poprawnie.


```
_id: ObjectId('64ede9bd62e66e0678a103e9')
▸ degrees: Array
▾ roles: Array
    0: "REGISTERED_USER"
▸ opinions: Array
  firstName: "Dominik"
  lastName: "Trojanowski"
  password: "$2b$10$/aU4vv8NBs.1fqm0Lxb9p.kc6ciK4cz89XSenUiVwx7AOAoB3KFCa"
  email: "dominik@gmail.com"
  gender: "MALE"
  createdAt: 2023-08-31T13:10:38.987+00:00
  updatedAt: 2023-08-31T13:10:38.987+00:00
  __v: 0
```


Rysunek 16 Dokument reprezentujący użytkownika z poziomu bazy danych

Rysunek nr 16 prezentuje efekt utworzenia nowego konta użytkownika. Następnie wywołano zapytanie dodające nowy temat pracy dyplomowej z *tokenem* użytkownika o podstawowych uprawnieniach.



Rysunek 17 Rezultat wysłania zapytania dodającego nową propozycję tematu pracy

Wysłanie zapytania dodającego nową propozycję tematu pracy zostało odrzucone. Został zwrócony *kod 401* informujący o braku autoryzacji w odniesieniu do konkretnego zasobu. Jest to oczekiwany rezultat. Test zakończył się sukcesem.

4. Test poprawności procesu rejestracji

Rejestracja jest jednym z podstawowych akcji wykonywanych w serwisach różnego typu. Jest to proces wymagający szczególnej weryfikacji zgodności z dokumentacją i ogólnymi standardami.

Celem testu jest weryfikacja poprawności procesu rejestracji użytkownika rozumiana jako zgodność z wymaganiami.

Wymagania stawiane względem rejestracji podczas testów:

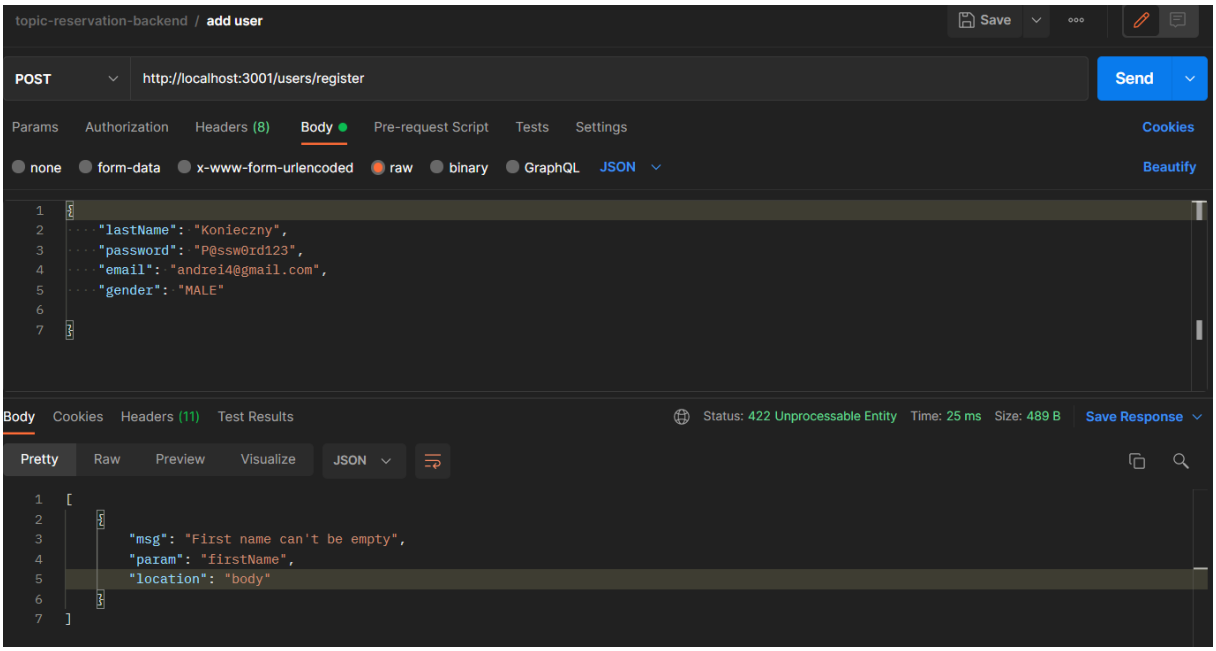
- użytkownik musi dostarczyć wszystkie niezbędne dane: imię, nazwisko, hasło, email i płeć,
- podany adres email musi mieć poprawną postać,
- hasło musi spełniać łącznie następujące wymagania:
  - minimalna długość na poziomie 8 znaków
  - maksymalna długość na poziomie 30 znaków
  - zawierać przynajmniej jedną cyfrę
  - zawierać przynajmniej jeden znak specjalny
  - zawierać przynajmniej jedną dużą i małą literę
- email użytkownika jest polem unikalnym,
- pole imię ma długość pomiędzy 2-30 znaków,
- pole nazwisko ma długość pomiędzy 2-30 znaków,
- płeć jest wartością wybraną spośród wartości: *MALE*, *FEMALE*, *OTHER.*

W ramach testu wykonano serię zapytań rejestracyjnych do *REST API*. Zapytania zostały wysłane za pośrednictwem narzędzia *Postman*. Celem testu było sprawdzenie czy pomimo pewnych braków zapytania zostaną poprawnie przetworzone. Rysunek nr 18 przedstawia zbiór wymaganych pól z przykładowymi wartościami niezbędnymi do zarejestrowania nowego użytkownika.

```
{
  "firstName": "Andrei",
  "lastName": "Konieczny",
  "password": "P@ssw0rd123",
  "email": "andrei2@gmail.com",
  "gender": "MALE"

}
```


Rysunek 18 Obiekt prezentujący przykładowe dane wymagane przy rejestracji

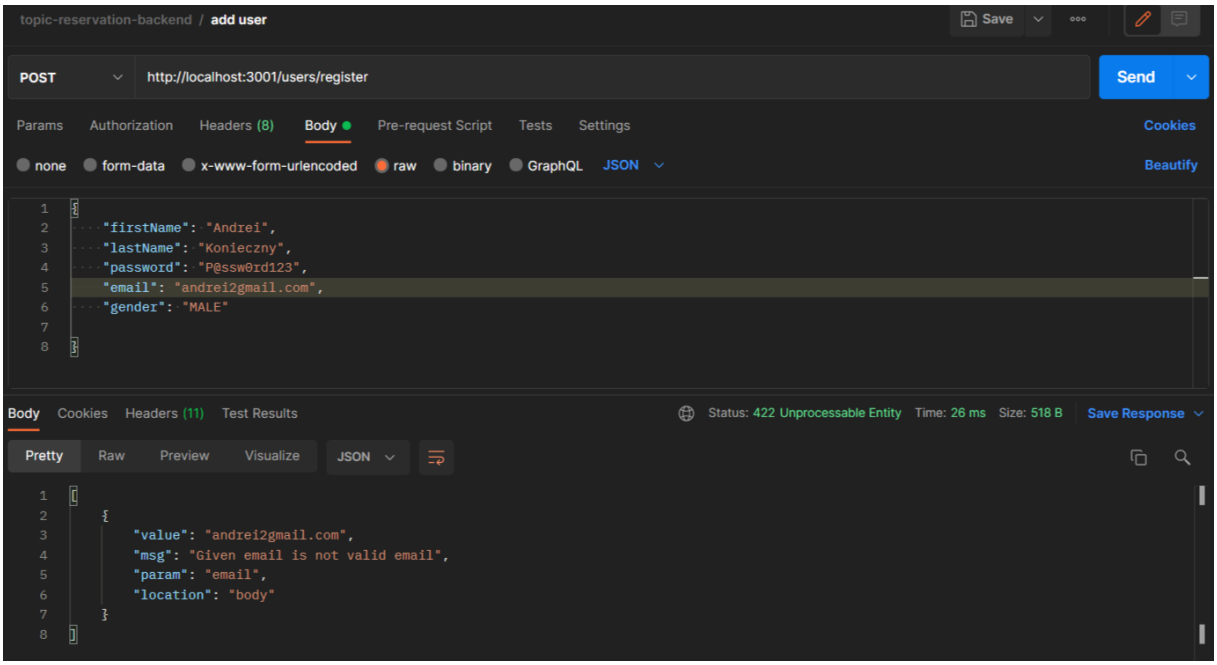
Rysunek nr 19 prezentuje odpowiedź serwera w przypadku kiedy nie przesłano wszystkich wymaganych pól.



Rysunek 19 Efekt zapytania rejestracyjnego przy braku danych

W przypadku pól *lastName*, *password*, *emai orazl gender* logika działa adekwatnie. Pozwala to stwierdzić, że walidacja ilościowa zawartości zapytania rejestracyjnego działa poprawnie.

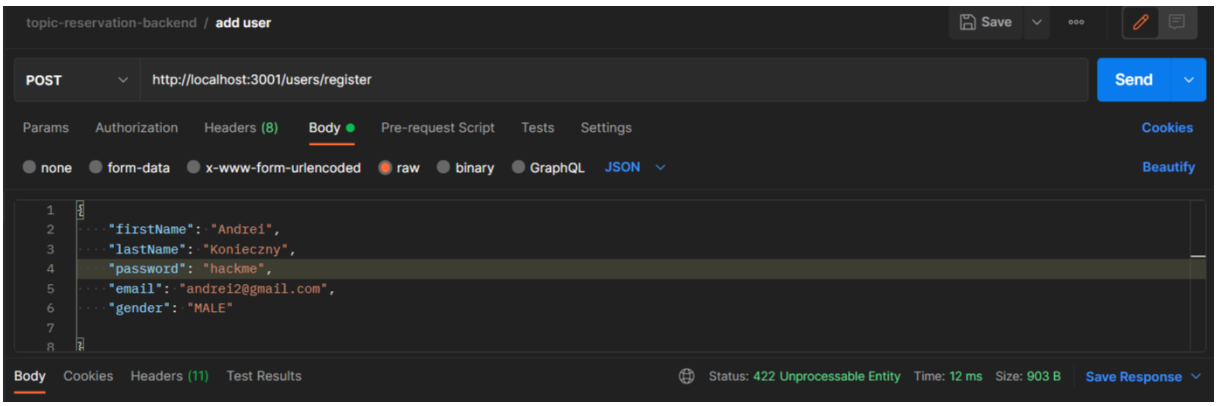
W następnym kroku sprawdzono czy wymagania jakościowe względem wprowadzanych danych również muszą zostać spełnione. W poniższym teście sprawdzono czy przesyłając niepoprawny adres email konto użytkownika zostanie pomyślnie utworzone.



Rysunek 20 Zapytanie wraz z odpowiedzią serwera przy podania błędnego adresu email

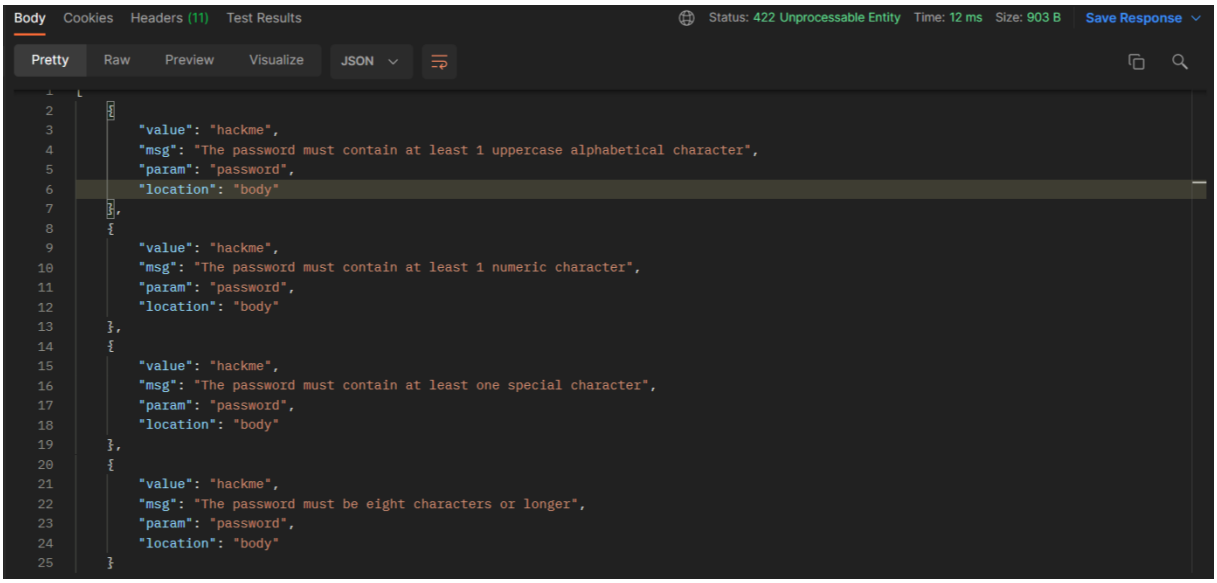
Rysunek nr 20 prezentuje odpowiedź serwera. Zwrócona została informacja o braku poprawności pola przechowującego adres email. Jest to efekt oczekiwany a test został zakończony sukcesem.

Przetestowano również wymagania dotyczące złożoności hasła. Przetestowano kilka wariantów haseł nie spełniających wymagań. Najciekawszym przypadkiem okazał się hasło składające się zaledwie z 6 znaków.



Rysunek 21 Zapytanie testujące wymagania złożoności hasła

Efektem zapytania była następująca odpowiedź serwera.

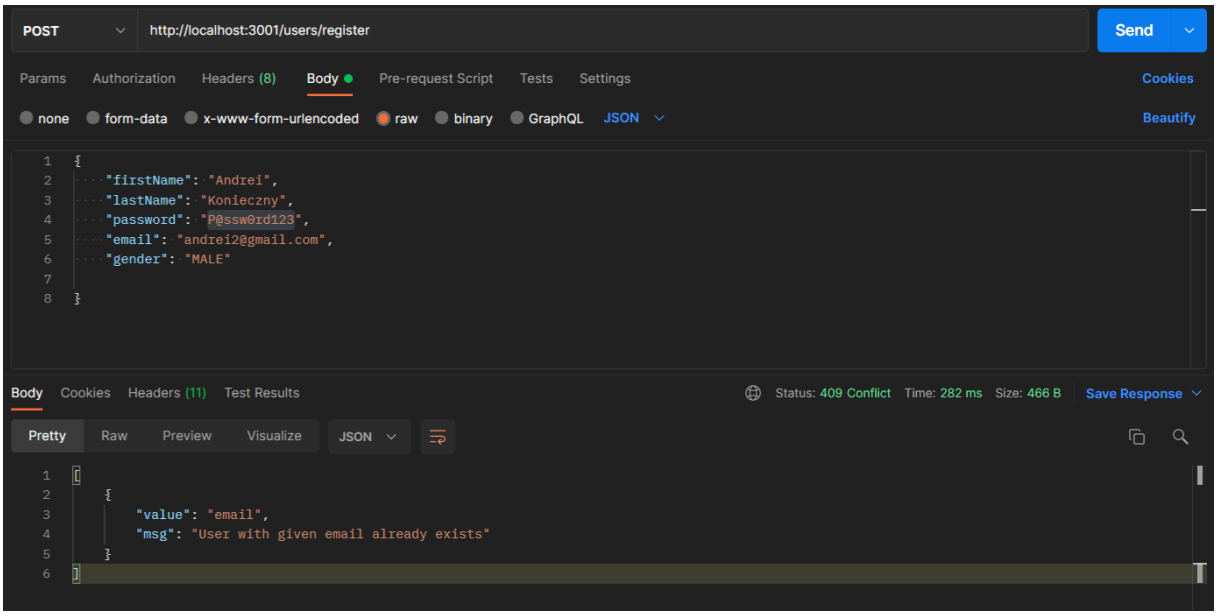


Rysunek 22 Odpowiedź serwera na zapytanie testujące wymagania złożoności hasła

Treść odpowiedzi serwera prezentowana na rysunku nr 22 zawiera 4 błędy dotyczące kolejno: braku wielkiej litery, braku znaku alfanumerycznego, braku znaku specjalnego i niespełnieniu wymagania długości hasła. Uzyskano oczekiwany efekt a test został zakończony sukcesem.

Przetestowano również przypadek dotyczy sytuacji kiedy użytkownik o podanym adresie email już istnieje. Pole email jest unikalne względem użytkownika.

Rysunek 23 Dokument reprezentujący użytkownika w bazie danych



Rysunek 24 Efekt próby utworzenia użytkownika w przypadku kiedy użytkownik o podanym adresie email już istnieje

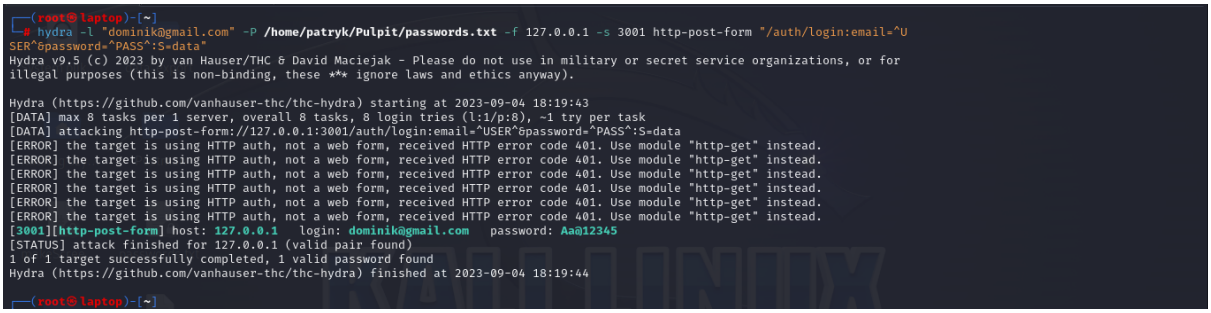
W efekcie serwer zwrócił odpowiedź z informacją o istnieniu użytkownika przypisanego do podanego adresu email. Uzyskano oczekiwany efekt. Test został zakończony sukcesem.

Sprawdzono również różne nieprawidłowe warianty dotyczące długości wartości pól *firstName*, *lastName* oraz wartości pola *gender*. Wszystkie przeprowadzone testy dały oczekiwany rezultat.

Przeprowadzone testy pozwalają stwierdzić, że proces rejestracji użytkownika spełnia wymagania ilościowe i jakościowe względem przekazywanych danych. Rejestracja wymaga od użytkownika podania unikalnego adresu email oraz hasła średniego poziomu złożoności. Uzyskane rezultaty pozwalają stwierdzić zgodność procesu rejestracji z wymaganiami.

5. Test odporności na atak Brute Force

Do przetestowania odporności na ataki typu *Brute Force* wykorzystano narzędzie hydra dostępne z poziomu systemu Kali Linux.



Rysunek 25 Rezultat próby złamania hasła użytkownika narzędziem hydra

Narzędzie uruchomione zostało z następującymi flagami:

- flaga *-l* oznacza login, który będzie testowany (można podać również flagę *-L* umożliwiającą wskazanie słownika zawierającego nazwy użytkowników, które mają zostać przetestowane),
- flaga *-P* umożliwia wskazanie słownika zawierającego hasła, które mają zostać przetestowane,

- flaga *-f* pozwala wskazać atakowany adres IP,
- flaga *-s* pozwala wskazać port.

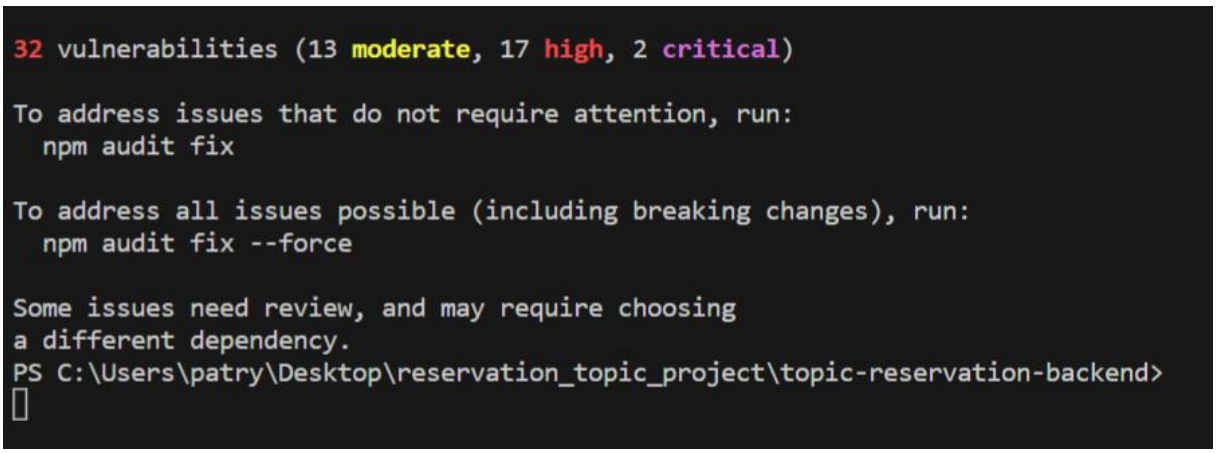
Rysunek 26 Zawartość pliku zawierającego listę testowanych haseł

Podając *http-post-form* wskazano jakie pod narzędzie ma zostać wykorzystane do procesu łamania hasła. Nazwa sugeruje sposób działania. W testowanym przypadku wysyłane będą żądania typu POST za pośrednictwem protokołu *HTTP*. Wykorzystując odpowiednią składnię zdefiniowano układ przesyłanych danych oraz wskazano jak rozumiany ma być sukces znalezienia prawidłowego hasła. Układ oczekiwanych danych sprawdzono poprzez wykonanie zapytania logowania przez część frontendową.

Rysunek nr 25 prezentuje efekt przeprowadzenia ataku. Łącznie podjęto 8 prób złamania hasła użytkownika, z których ostatnia zakończyła się powodzeniem. Każda kolejna próba była rozpatrywana przez serwer na tych samych zasadach co pierwsza z nich. Brak sensownego ograniczenia ilości prób logowania umożliwia przeprowadzenie skutecznego ataku typu *Brute Force.* Przeprowadzony test został niezaliczony i wymaga pilnej interwencji.

6. Weryfikacja podatności zewnętrznych zależności

Obecnie projekty w dużej mierze korzystają z zewnętrznych bibliotek. Znacząco przyspiesza to proces tworzenia rozwiązań. Wiąże się jednak ze wzrostem ilości podatności. Im więcej zewnętrznych zależności tym większa szansa, że tworzone rozwiązanie posiada luki związane z wykorzystaniem zewnętrznego kodu. W celu weryfikacji podatności w użytych bibliotekach zewnętrznych zostanie wykorzystane narzędzie *npm audit*.

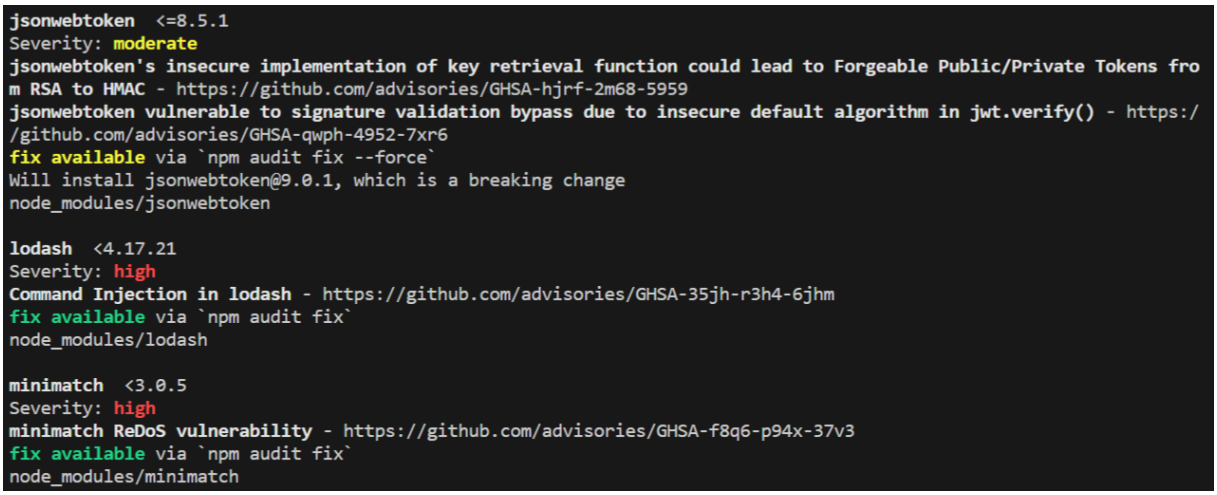


Rysunek 27 Rezultat wykonania polecenia npm audit w repozytorium backendu

W wyniku uruchomienia komendy polecenia zwrócone zostało zestawienie wykrytych podatności w wykorzystywanych obecnie wersjach zależności. Ujawnione zostały łącznie 32 podatności. Dwie mają charakter krytyczny, 17 ma charakter poważny a 13 charakter umiarkowany. Wszystkie podatności powinny zostać poddane analizie i jeżeli to możliwe wyeliminowane. W pierwszej kolejności należy wyeliminować podatności zakwalifikowane grupy krytycznej.

Rysunek 28 Wybrane krytyczne podatność części backendowej

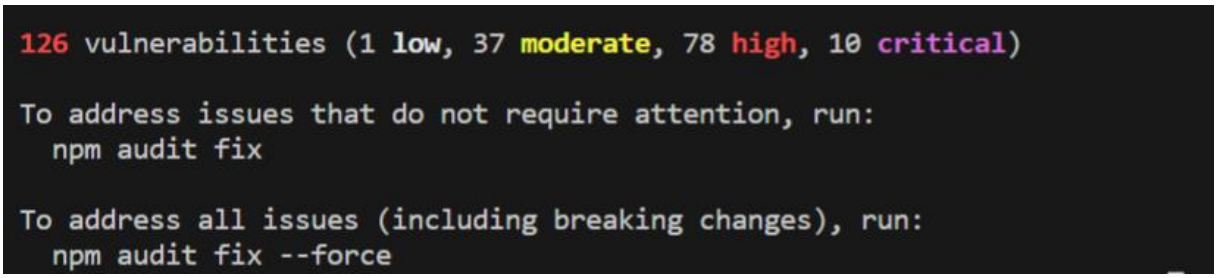
Podatności widoczne na rysunku nr 28 zakwalifikowane zostały do grupy krytycznych. Dotyczą zanieczyszczenia prototypów. Jest to luka w języku JavaScript umożliwiająca dodanie dowolnych właściwości do prototypów obiektów globalnych. Obiekty zdefiniowane przez użytkownika mogą odziedziczyć nadmiarowe pola. Źródłem podatności są określone wersje bibliotek *minimist* oraz *mongoose*. Większość zagrożeń zawiera hiperłącze do wątku opisującego szerzej problematykę danej podatności. W badanym przypadku bardziej ciekawe okazały się podatności zaliczone do poziomu wysokiego i umiarkowanego.



Rysunek 29 Wybrane poważne i umiarkowane podatność części backendowej

Rysunek nr 29 prezentuje trzy interesujące podatności. Pierwsza z nich dotyczy biblioteki *jsonwebtoken*. Zaliczona została do grupy umiarkowanej. Podatność ma jednak spore znaczenie, ponieważ może prowadzić do fałszowania *tokenów* w odniesieniu do odzyskiwania klucza z *RSA* do *HMAC*. Druga z podatności została wykryta w bibliotece *lodash*. Dokładniej w wersji 4.17.21. Wykryta podatność dotyczy możliwości wstrzykiwania komend. Ostatnia z podatności tyczy się biblioteki *minimatch*. Tym razem wykryta luka dotyczy podatności na ataki typu *RedoS*.

Rysunek nr 30 przedstawia wynik wykonania polecenia *npm audit* w repozytorium frontendu.



Rysunek 30 Wynik wywołania komendy npm audit w repozytorium frontendu

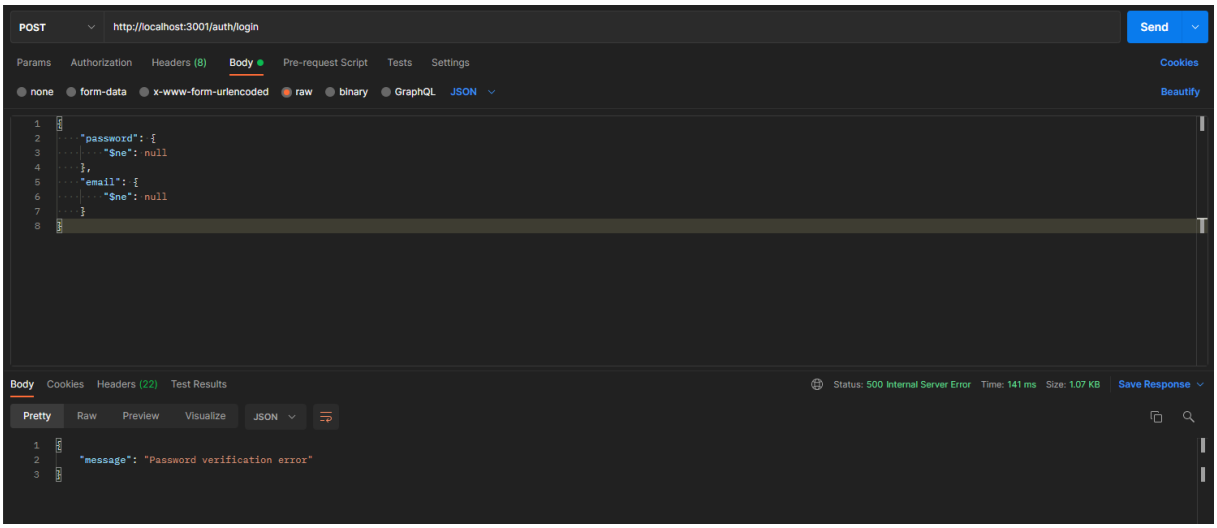
Łącznie wykryto 126 podatności. Zwiększona ilość w porównaniu do backendu wynika bezpośrednio ze znaczniej większej ilości bibliotek użytych po stronie frontendu.

Wykryte błędy krytyczne dotyczą m. in.:

- niepoprawnej nazwy hosta w przypadku biblioteki *url-pars,*
- podatność na zanieczyszczenie prototypami w przypadku bibliotek: *json-sche*ma, *merge-deep*, *property-expr* oraz *minimist,*
- podatność na ataki typu *ReDoS* w przypadku *loader-utils,*
- podatność na niewłaściwą neutralizację znaków specjalnych użytych w cudzysłowie przypadku biblioteki *shell-quote.*

7. Test odporności na ataki typu NoSQL

Test odporności na ataki typu NoSQL przeprowadzono w odniesieniu do procesu logowania użytkownika. Celem ataku było ominięcie procesu logowania poprzez wstrzyknięcie wyrażenia, który pozwoli uzyskać token jednego z użytkowników bez podawania prawidłowych danych logowania. W tym celu specjalnie spreparowano treść zapytania. Jako wartości pól password oraz email podano obiekt typu JSON. Obiekt ten zawiera parę klucz -wartość. W obu przypadkach kluczem jest ciąg znaków „$ne” będący operatorem w MongoDB oznaczającym dosłownie „nie równe”. Wartością klucza „$ne” jest *null*. Po zinterpretowaniu przez silnik bazy danych powinny zostać zwrócone obiekty, których hasło nie jest puste. Zakładając, że każdy z utworzonych użytkowników ma przypisane hasło zostanie zwrócony cała kolekcja obiektów. Z dużym prawdopodobieństwem w przypadku logowania wykorzystana została funkcja *findOne*, która zwróci pierwszy obiekt spełniający warunki. Tym samym oczekuję się, że zostanie zwrócony obiekt pierwszego użytkownika w bazie danych.



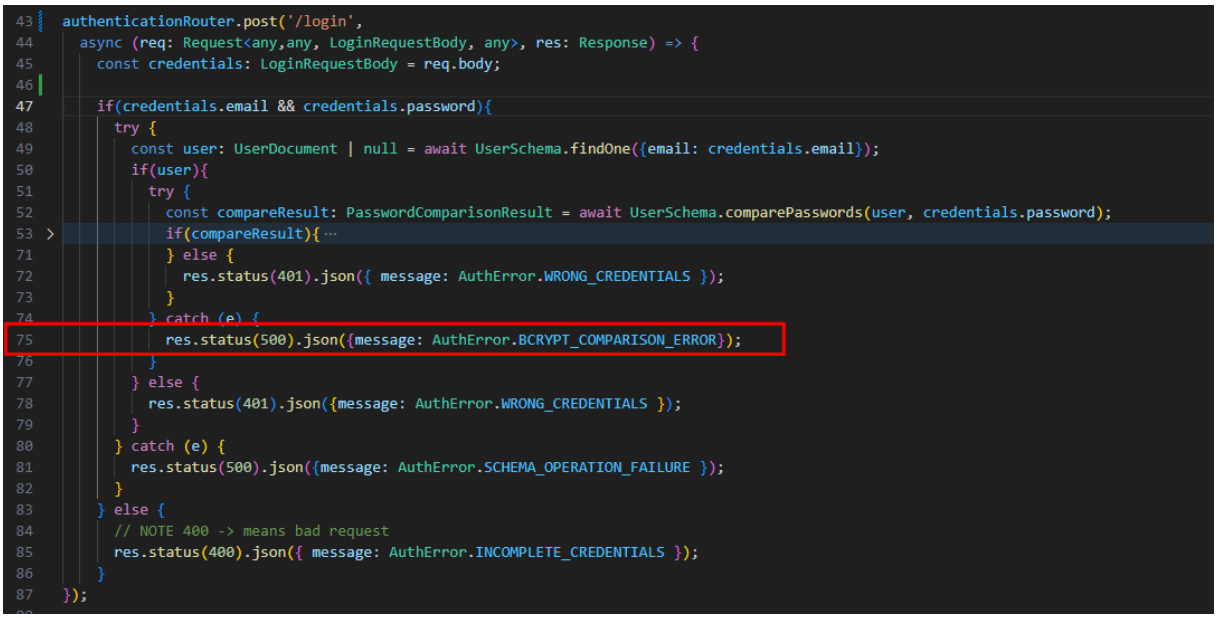
Rysunek 31 Zapytanie testujące odporność na atak NoSQL Injection

Jak widać w wyniki przesłania takiego zapytania otrzymaliśmy kod *500* sugerujący błąd po stronie serwera. Uzyskany rezultat wymaga sprawdzenia kodu części backendowej.

```
export enum AuthError {
  SCHEMA_OPERATION_FAILURE = "Operation on user schema failure",
  WRONG_CREDENTIALS = "Wrong credentials",
  INCOMPLETE_CREDENTIALS = 'Incomplete credentials',
  BCRYPT_COMPARISON_ERROR = 'Password verification error',
  INVALID_TOKEN = 'Invalid token',
  TOKENS_NOT_INCLUDED = 'Tokens not included',
  USER_NOT_FOUND = 'User not found',
  WRONG_TOKEN_PAYLOAD = 'Wrong token payload',
  FIND_USER_OPERATION_FAIL = 'Find user operation fail',
  INVALID_REFRESH_TOKEN = 'Invalid refresh token'
}
```

Rysunek 32 Typ wyliczeniowy możliwych odpowiedzi serwera na zapytanie pozwalające zalogować użytkownika

Rysunek nr 32 przedstawia możliwe odpowiedzi serwera na zapytanie pozwalające zalogować użytkownika. Treść wiadomości otrzymanej odnosi się do klucza *BCRYPT_COMPARISON_ERROR*. Skorelowanie wartości odpowiedzi z wartością klucza pozwoli zidentyfikować miejsce, do którego kod został wykonany.



Rysunek 33 Fragment kodu odpowiedzialny za logowanie użytkownika

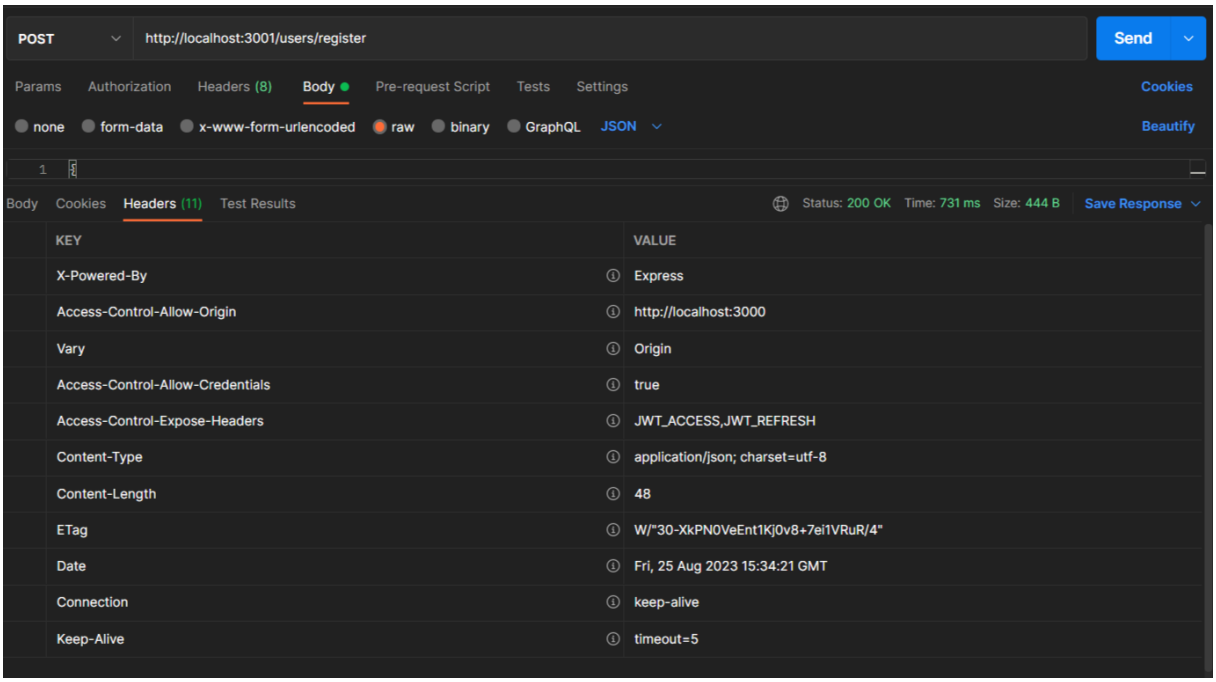
Fragment kodu przedstawiony na rysunku nr 33 pokazuje, że atak częściowo się powiódł. Dane odczytane przez backend zostały poprawnie zinterpretowane przez silnik bazy danych w wyniku czego został zwrócony pierwszy obiekt z bazy danych. Następnie podjęto próbę weryfikacji podanego hasła z odnalezionym użytkownikiem. W tym celu została wykorzystana logika udostępniona przez bibliotekę *bcrypt* pozwalającej na konwersję hasła do tzn. *hasha*. *Hash* jest formą przechowywania hasła, które zostało odpowiednio przetworzone za pomocą określonych algorytmów kryptograficznych. Błąd został uchwycony dopiero na etapie porównywania podanego hasła z odszyfrowanym hasłem z bazy danych. Oczekiwano, że podanie błędnych wartości nie spowoduje wykonania kodu na tym poziomie. Mimo iż atak się w pełni nie powiódł test zakończył się niepowodzeniem i będzie wymagane podjęcie akcji w tym obszarze.

8. Weryfikacja poprawności konfiguracji serwera

W tej sekcji sprawdzono wpływ konfiguracji części backendowej na kwestie bezpieczeństwa.

Pierwszym etapem weryfikacji konfiguracji było sprawdzenie protokołu wykorzystywanego do przesyłania danych. Serwer został skonfigurowany do komunikacji za pośrednictwem protokołu *HTTP* co niestety nie jest bezpiecznym sposobem przesyłu informacji. Związane jest to z przesyłaniem danych jawnym tekstem co naraża rozwiązanie na ataki typu *Man-in-the-middle*. W tym obszarze konfiguracja będzie musiała ulec zmianie.

Kolejnym etapem było sprawdzenie ilości oraz zawartości nagłówków odpowiedzi zwracanych przez serwer. W celu przeprowadzenia testu zwracanych nagłówków wykonano jedno z zapytań nie wymagających autoryzacji.



Rysunek 34 Nagłówki zwrócone przez serwer po wysłaniu zapytania rejestracyjnego

Rysunek nr 33 pokazuje nagłówki odpowiedzi serwera. Zwrócić uwagę należy na obecność nagłówka *X-Powered-By* w odpowiedzi serwera. Obecność tego nagłówka jest jedną z prostych form ujawnienia technologii wykorzystywanej przez rozwiązanie. W tym wypadku wartość odzwierciedla nazwę biblioteki wykorzystanej do stworzenia *REST API* w *Node.js*. Dodatkowo należy zauważyć brak nagłówków takich jak: *Content-Security-Policy*, *Cross-Origin-Resource-Policy*, *Cross-Origin-Resource-Policy*. W tym przypadku test nie został zaliczony i będzie wymagane podjęcia akcji.

Sprawdzono również reguły konfiguracji *CORS (ang. Cross Origin Resource Sharing)*. Polityka *CORS* pozwala złagodzić politykę *SOP (ang. Same Origin Policy)* dzięki czemu komunikacja między różnymi źródłami będzie możliwa. Rysunek nr 35 prezentuje fragment konfiguracji serwera dotyczący polityki *CORS*. W sekcji *origin* wskazano dwa źródła, które będą mogły komunikować się z serwerem. Wymienione źródła dotyczą adresów pod którymi działa frontendowa część systemu. W celu umożliwienia przesyłu ciasteczek do pola *credentials* przypisano wartość *true*. W konfiguracji wyszczególnione zostały niestandardowe nagłówki zawierające *tokeny*. Zdefiniowano również akceptowane metody w odniesieniu do zapytań.

```
app.use(cors({
  // NOTE below setting allow to transfer cookies over cors
  credentials: true,
  // NOTE below setting allow to make requests to server only from localhost
  origin: ['http://localhost:3000', 'http://127.0.0.1:3000'],
  // NOTE it is necessary to make JWT_ACCESS and JWT_REFRESH visible inside frontend request headers
  exposedHeaders: [COOKIES_WITH_TOKENS.JWT_ACCESS, COOKIES_WITH_TOKENS.JWT_REFRESH],
  methods: ['GET','POST','DELETE','UPDATE','PUT','PATCH']
}));
```

Rysunek 35 Konfiguracja CORS

Wykorzystywana konfiguracja nie budzi zastrzeżeń. Ten test został zakończony sukcesem.

4. Działania zwiększające bezpieczeństwo

1. Eliminacja zagrożeń dotyczących konfiguracji serwera

Największą modyfikacja w konfiguracji serwera dotyczy protokołu wykorzystywanego do przesyłania danych między klientem a serwerem. Protokół *HTTP* został zastąpiony protokołem *HTTPS* oferującym szyfrowanie. Wykorzystywanie protokołu *HTTPS* zapewnia, że cały ruch wysyłany i odbierany przez przeglądarkę jest szyfrowany. W praktyce oznacza to, że wszystkie dane poufne jak np. hasła czy dane osobiste będą trudne do przechwycenia. Dzięki wykorzystaniu protokołu *HTTPS* lokalna konfiguracja serwera bardziej przypomina konfigurację produkcyjną.

Do osiągnięcia celu w lokalnym środowisku wykonane zostały następujące kroki:

1. Wygenerowano 2048 bitowy klucz prywatny *RSA.*
2. Wygenerowano żądanie obsługi certyfikatu *CSR.*
3. Wygenerowano certyfikat *SSL* .
4. Podpisano certyfikat wcześniej wygenerowanym kluczem prywatnym.

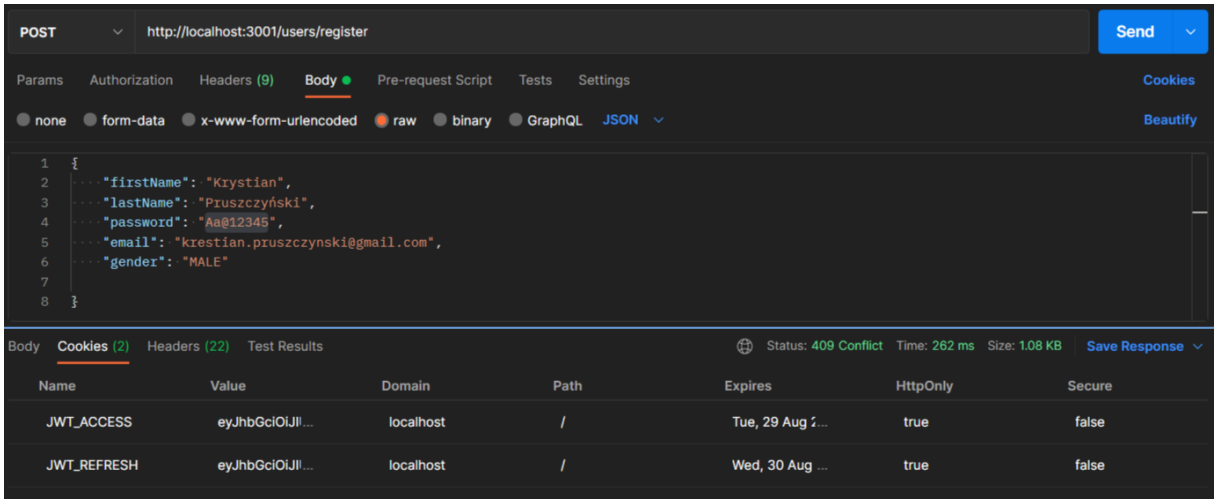
```
patry@DESKTOP-EBGNQNE MINGW64 ~/Desktop/reservation_topic_project/topic-reservat
ion-backend (master)
$ openssl genrsa -out key.pem
Generating RSA private key, 2048 bit long modulus (2 primes)
.....................+++++
.....................................................+++++
e is 65537 (0x010001)

patry@DESKTOP-EBGNQNE MINGW64 ~/Desktop/reservation_topic_project/topic-reservat
ion-backend (master)
$ openssl req -new -key key.pem -out csr.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:PL
State or Province Name (full name) [Some-State]:Malopolska
Locality Name (eg, city) []:Krakow
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:

patry@DESKTOP-EBGNQNE MINGW64 ~/Desktop/reservation_topic_project/topic-reservat
ion-backend (master)
$ openssl x509 -req -days 9999 -in csr.pem -signkey key.pem -out cert.pem
Signature ok
subject=C = PL, ST = Malopolska, L = Krakow, O = Internet Widgits Pty Ltd
Getting Private key
```

Rysunek 36 Proces stworzenia nowego certyfikatu SSL

Rysunek 36 przedstawia kolejno realizowane kroki w odniesieniu do certyfikatu.

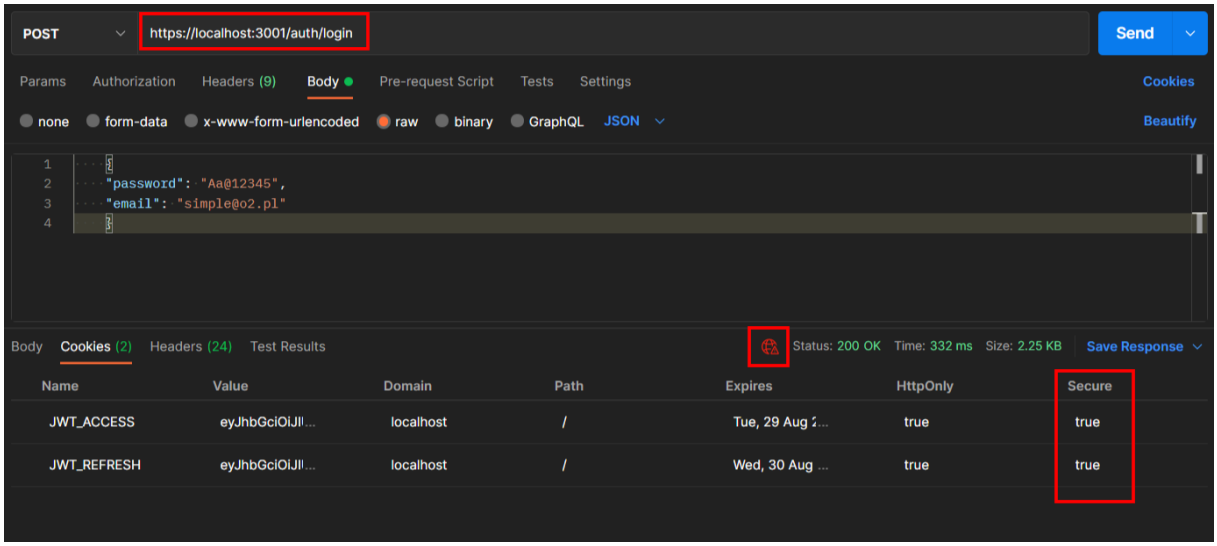
Wykorzystanie szyfrowanego protokołu do komunikacji pozwoliło rozwiązać problem wykrytej podatność dotyczącej ciasteczek przechowujących *tokeny*. W przypadku ciasteczek: *JWT_ACCESS* oraz *JWT_REFRESH* wartość flagi *Secure* ustawiona była na

wartość *false*. Ustawienie wartość flagi *Secure* na *true* w sytuacji kiedy serwer ciągle wykorzystywał protokół *HTTPS* uniemożliwiało przesyłanie ciasteczek. Tym samym wszystkie działania wymagające autoryzacji w aplikacji nie były możliwe. Flaga *Secure* gwarantuje, że pliki cookie przesyłane są wyłącznie za pośrednictwem protokołu *HTTPS* gwarantującego szyfrowanie przesyłanych danych.



Rysunek 37 Zapytanie pozwalające zalogować użytkownika do serwisu przesłane za pośrednictwem HTTP

Rysunek nr 37 prezentuje ciasteczka zwracane przez serwer. W przypadku obu ciasteczek flaga *Secure* ma wartość *false*. Należy zwrócić uwagę, że powyższe zapytanie zostało wysłane za pomocą protokołu *HTTP*.

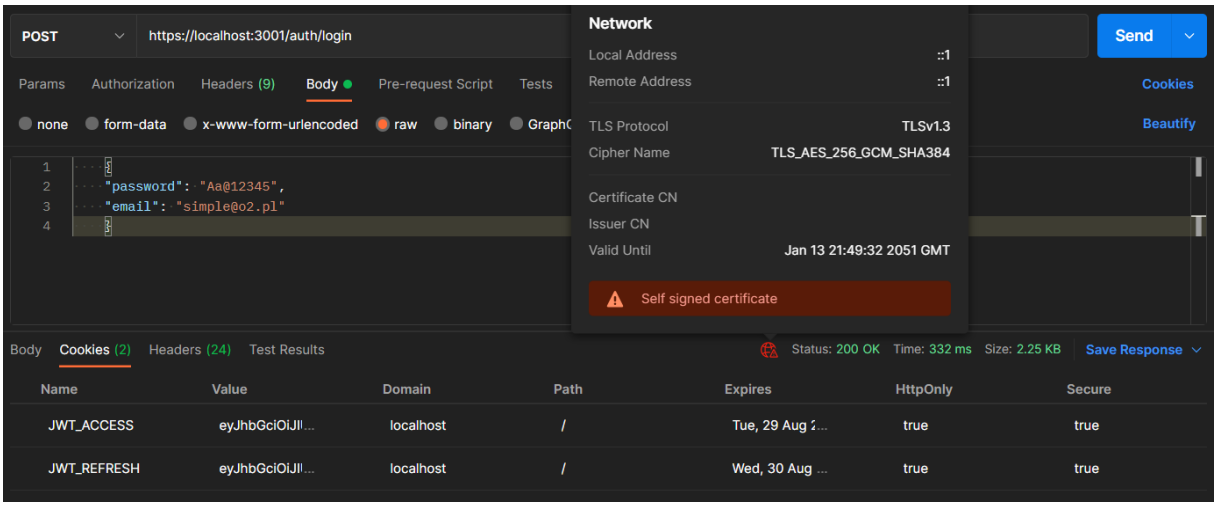


Rysunek 38 Zapytanie pozwalające zalogować użytkownika do serwisu przesłane za pośrednictwem HTTPS

Po odpowiedniej zmianie konfiguracji serwera komunikacja pomiędzy klientem a serwerem odbywa się za pośrednictwem protokołu *HTTPS*. Następnie w odniesieniu do ciasteczek wymuszono przesył wyłącznie za pomocą protokołu *HTTPS*. W odpowiedzi serwer zwrócił ciasteczka, których wartość flagi *Secure* ustawiona została na wartość *true* co potwierdza poprawność wprowadzonej konfiguracji. Efekt prezentowany jest na rysunku nr 38.

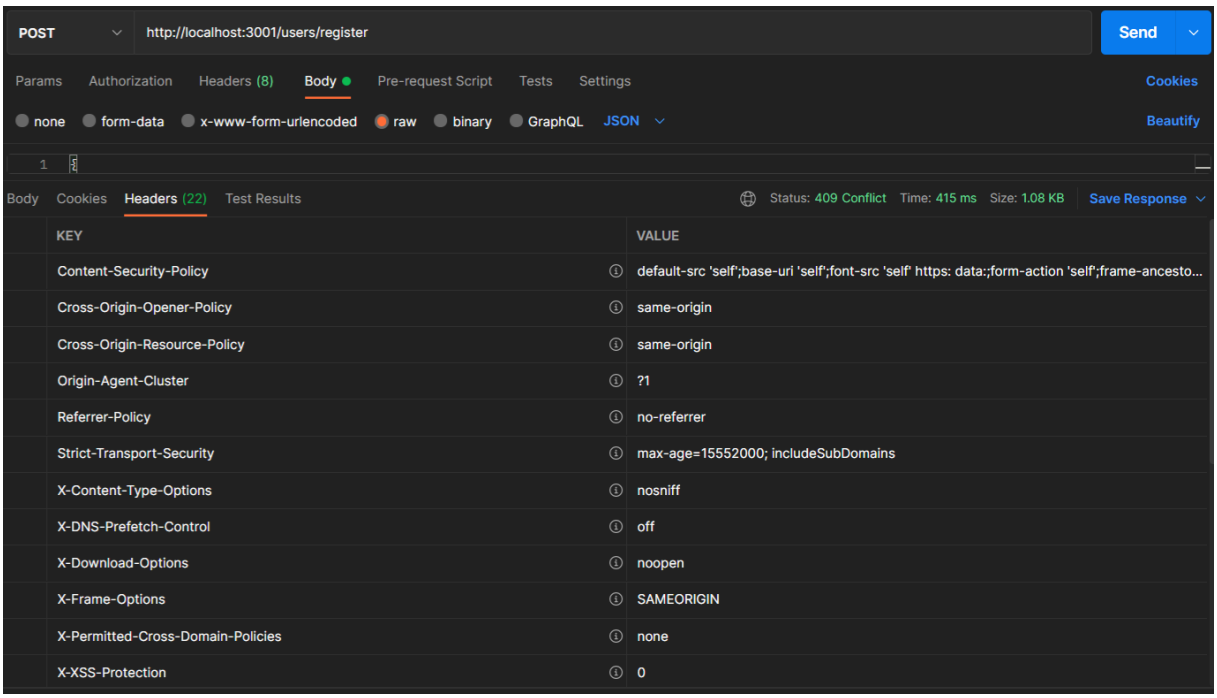
Rysunek nr 39 prezentuje treść ostrzeżenia dotycząca certyfikatu, który został podpisany „osobiście” przez autora. Certyfikaty uważane za bezpieczne podpisane są przez specjalne

urzędy certyfikacji. Jest to powodem pojawienie się stosownego ostrzeżenia w programie *Postman* Takie rozwiązanie przyjęto jedynie na potrzeby lokalnego środowiska. W środowisku produkcyjnym należy wykupić odpowiedni certyfikat *SSL* wystawiony przez jednego z dostawców certyfikatów.



Rysunek 39 Ostrzeżenie dotyczące certyfikatu

Na tym jednak nie kończą się kwestie dotyczące konfiguracji serwera. W celu stworzenia pełniejszej polityki bezpieczeństwa dotyczącej nagłówków została wykorzystana biblioteka *helmet*. Wybrana bibliotek pozwoliła ustawić domyślne wartości dla nagłówków mających szczególnie istotne znaczenie pod kątem bezpieczeństwa. Rysunek nr 40 prezentuje nagłówki odpowiedzi serwera po wykorzystaniu biblioteki *helmet* z domyślną konfiguracją.



Rysunek 40 Efekt wykonania zapytania rejestracyjnego wykorzystaniu biblioteki helmet z domyślną konfiguracją

Po wykorzystaniu biblioteki ilość nagłówków odpowiedzi serwera zwiększyła się dwukrotnie z poziomu 11 do 22. W efekcie usunięty został również nagłówek *X-Powered-By*. Dzięki temu rozpoznanie wykorzystanej technologii zostało znacząco utrudnione.

Powszechnie występującą praktyką jest również wskazywania nieprawidłowej wartości nagłówka *X-Powered-By* w celu wprowadzenia w błąd atakującego.

Wykorzystanie biblioteki *helmet* spowodowało dodanie nagłówków *Cross-Origin-Resource-Policy* i *Cross-Origin-Opener-Policy*. Zawartość wymienionych nagłówków została ustawiona na wartość *same-origin*.

Ustawienie wartości nagłówka *Cross-Origin-Resource-Policy* na wartość *same-origin* pozwala zabezpieczyć obrazki przed atakami *Spectre* i kompromisowym renderowaniem. Nagłówek z tą wartością powoduje, że zasób będzie mógł zostać załadowany wyłącznie jeżeli odwołuje się do tego samego źródła. Proces weryfikacji leży po stronie mechanizmu przeglądarki.

Nagłówek *Cross-Origin-Opener-Policy* reguluje kwestię udostępniania grupy kontekstów przeglądania przez dokumenty najwyższego poziomu względem dokumentów pochodzących z innych źródeł. Ustawienie wartości tego nagłówka na wartość *same-origin* izoluje kontekst przeglądania, który dostępny jest wyłącznie dla dokumentów tego samego pochodzenia.

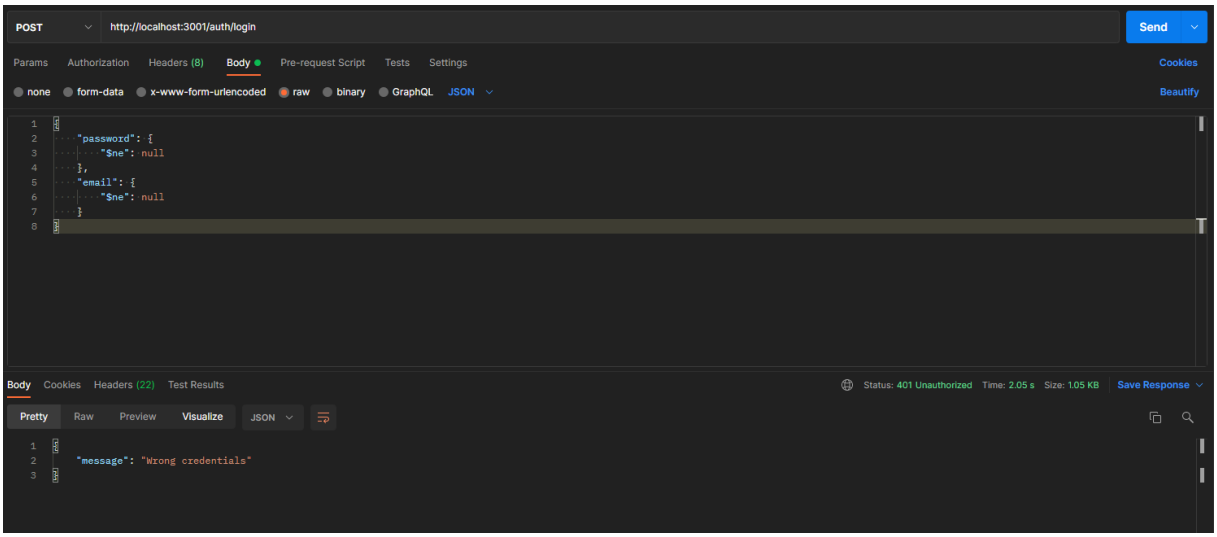
2. Eliminacja podatności atakiem NoSQL Injection

W celu eliminacji wykrytego zagrożenie zostanie wykorzystany mechanizm sanityzacji dostępny w bibliotece *mongoose* od wersji 6.

```
43  authenticationRouter.post('/login',
44    async (req: Request<any,any, LoginRequestBody, any>, res: Response) => {
45      const credentials: LoginRequestBody = req.body;
46
47      if(credentials.email && credentials.password){
48        try {
49          const user: UserDocument | null = await UserSchema.findOne({email: credentials.email}).setOptions({
50            sanitizeFilter: true,
51          });
52
53          if(user){
54            try {
55              const compareResult: PasswordComparisonResult = await UserSchema.comparePasswords(user, credentials.password);
56              if(compareResult){ ···
74              } else {
75                res.status(401).json({ message: AuthError.WRONG_CREDENTIALS });
76              }
77            } catch (e) {
78              res.status(500).json({message: AuthError.BCRYPT_COMPARISON_ERROR});
79            }
80          } else {
81            res.status(401).json({message: AuthError.WRONG_CREDENTIALS });
82          }
83        } catch (e) {
84          res.status(500).json({message: AuthError.SCHEMA_OPERATION_FAILURE });
85        }
86      } else {
87        // NOTE 400 -> means bad request
88        res.status(400).json({ message: AuthError.INCOMPLETE_CREDENTIALS });
89      }
90  });
```

Rysunek 41 Implementacja sanityzacji w odniesieniu do danych logowania

Rysunek nr 41 prezentuje zmodyfikowany kod. Wykorzystany został mechanizm sanityzacji dostępny bezpośrednio w bibliotece *mongoose* w wersji 6. W przypadku niższej wersji można wykorzystać również bibliotekę *express-mongo-sanitize*. Po wprowadzeniu stosownych zmian błąd zwracany jest na wcześniejszym etapie co zostało zaprezentowane na rysunku nr 42. Jest to efekt oczekiwany.



Rysunek 42 Odpowiedź serwera po wprowadzeniu sanityzacji danych

3. Eliminacja zagrożeń atakiem Cross Site Scripting

W celu eliminacji zagrożeń atakiem typu *XSS* przeprowadzono szereg działań.

Pierwszym z etapów było stworzenie polityki *Content Security Policy* poprzez ustawienie odpowiedniej wartości nagłówka *Content-Security-Policy* odpowiedzi serwera.

Zastosowano następującą politykę:

default-src 'self';base-uri 'self';font-src 'self' https: data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests

Znaczenie poszczególnych dyrektyw:

- dyrektywa *default-src* wykorzystywana jest w przypadku kiedy szczegółowe dyrektywy dotyczące zasobu nie zostały zdefiniowane,
- dyrektywa *base-uri* definiuje do jakiego źródła może odwoływać się element *<base>,*
- dyrektywa *font-src* definiuje z jakich źródeł będą mogły zostać pobrane czcionki,
- dyrektywa *form-action* pozwala określić pule adresów, do których będą przesyłane formularze,
- dyrektywa *frame-ancestors* pozwala określać prawidłowe elementy nadrzędne pozwalające na osadzenie strony,
- dyrektywa *img-src* pozwala zdefiniować pulę adresów z których będą mogły zostać załadowane pliki graficzne,
- dyrektywa *object-src* określa źródła dla elementów *<applet> <embed>* i *<object>,*
- dyrektywa *script-src* pozwala określić pulę adresów, z których będą mogły zostać załadowane skrypty,
- dyrektywa *script-src-attr* pozwala określić prawidłowe źródła dla procedur obsługi zdarzeń definiowanych w sposób liniowy,
- dyrektywa *style-src* pozwala określić pulę adresów, z których będą mogły zostać załadowane style,
- dyrektywa *upgrade-insecure-requests* pozwala wymusić pobieranie zasobów w oparciu o protokół *HTTPS.*

Większość wartości tych dyrektyw została ustawiona na wartość *self*. W odniesieniu do zasobów pozwala na ich załadowanie wyłącznie w przypadku kiedy spełniona została zasada tego samego źródła. Zasada wymaga zgodności na poziomie protokołu, hosta oraz portu. W celu pełniejszego zrozumienia w tabeli nr 1 zaprezentowano porównanie zgodności źródła kilku adresów URL w odniesieniu do adresu https://google.com

Tabela 1 Tabela zgodności adresów URL ze wskazanym źródłem

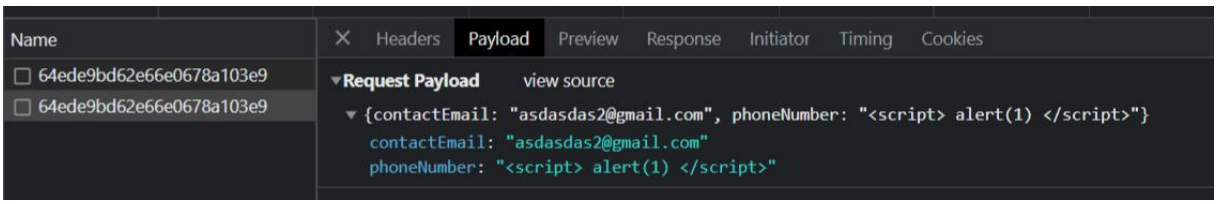
URL	Wynik	Powód
http://google.com	Brak zgodności	Wykorzystanie odmiennego protokołu
https://google.com:108	Brak zgodności	Wykorzystanie odmiennego portu
https://google.com/about	Zgodność	-

Kolejnym etapem było uzupełnienie braków w walidacji danych wprowadzanych przez użytkownika po stronie frontendu. W tym celu wykorzystana została biblioteka *Yup*. Bardziej istotną kwestią okazało się zaimplementowanie walidacja i sanityzacja danych po stronie backendowej. Kod realizujący analizowaną logikę widoczny jest na rysunku nr 43.

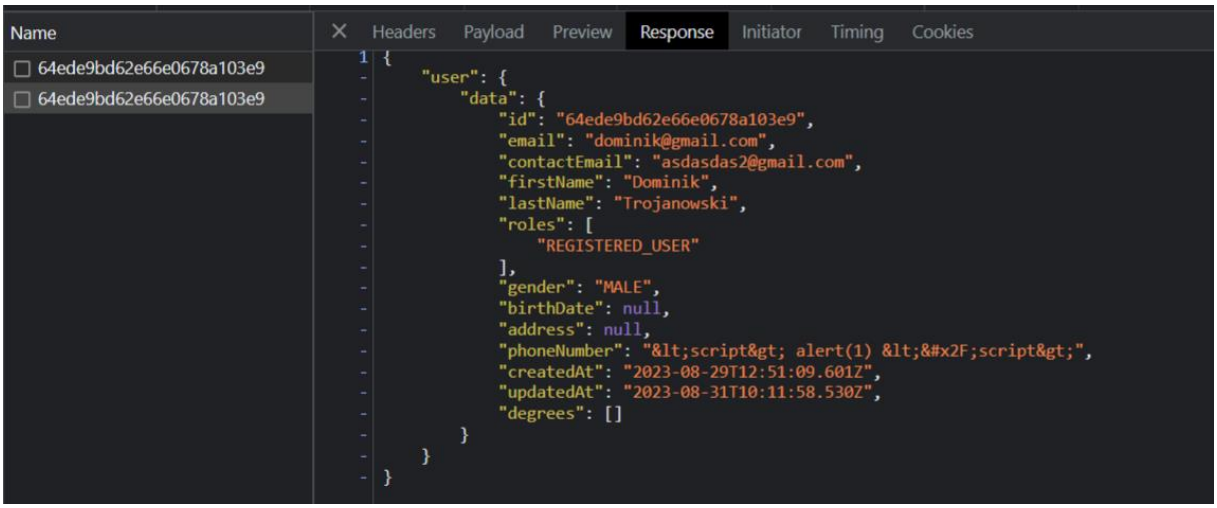
```
export const contactDataValidator = (): ValidationChain[] => [
  check('contactEmail')
    .isEmail(),
  check('phoneNumber')
    .escape()
]
```

Rysunek 43 Fragment kodu odpowiedzialny za walidację i sanityzację danych kontaktowych użytkownika

Po wprowadzeniu zmian przetestowano rozwiązanie. W tym celu przesłano do serwera zapytanie zawierające złośliwy fragment kodu w polu *phoneNumber* po uprzednim wyłączeniu walidacji po stronie frontendowej.



Rysunek 44 Zawartość zapytania aktualizującego dane kontaktowe użytkownika zawierające fragment złośliwego kodu



Rysunek 45 Odpowiedź serwera na zapytanie aktualizującego dane kontaktowe użytkownika zawierające fragment złośliwego kodu

Odpowiedź serwera prezentowana na rysunku nr 45 potwierdza poprawność działania walidacji i sanityzacji danych kontaktowych. W pierwszym kroku sprawdzono poprawność przekazanego adresu email. Ten został podany w poprawnej formie. Następnie w drugim etapie przeprowadzono proces sanityzacji tekstu wprowadzonego jako numer telefonu.

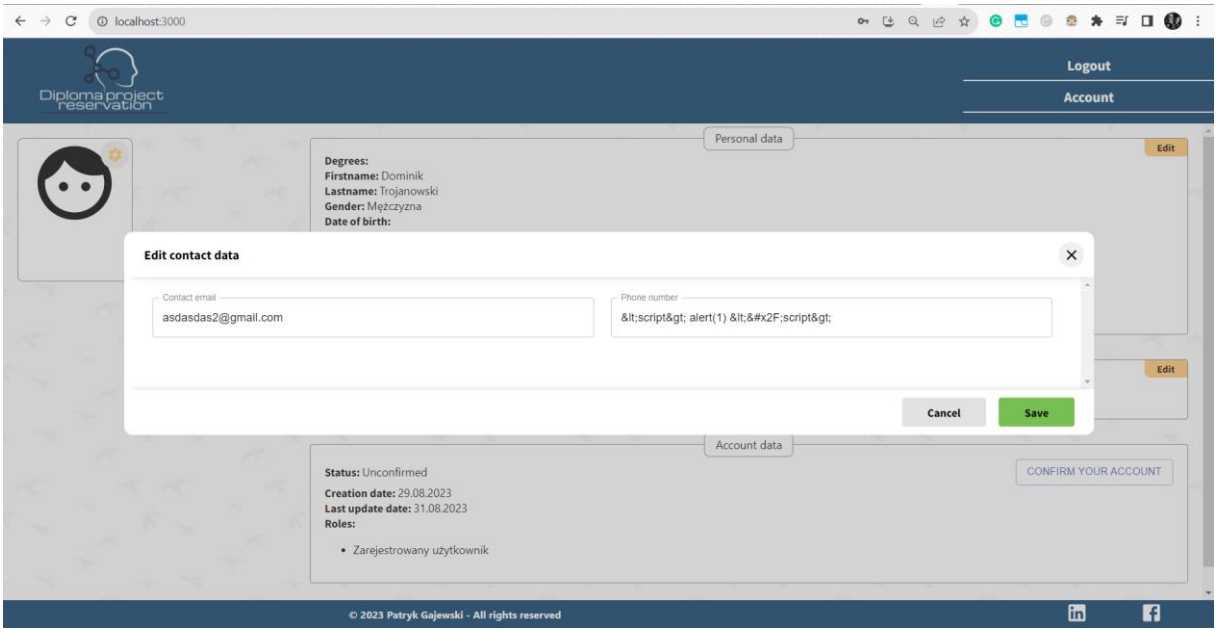
Z tekstu wejściowego:

<script> alert(1) </script>

Po sanityzacji z wykorzystaniem funkcji *escape()* otrzymano:

<script> alert(1) </script>

Konwersji poddane zostały znaki „<”, „>” oraz „/”. W ramach sanityzacji zostały odpowiednio zastąpione jako „<”, „&#x2F” oraz „>”. Dzięki temu tekst uruchamiany po stronie przeglądarki nie powoduje otwarcia okna informacyjnego.



Rysunek 46 Efekt zaimplementowana sanityzacji danych

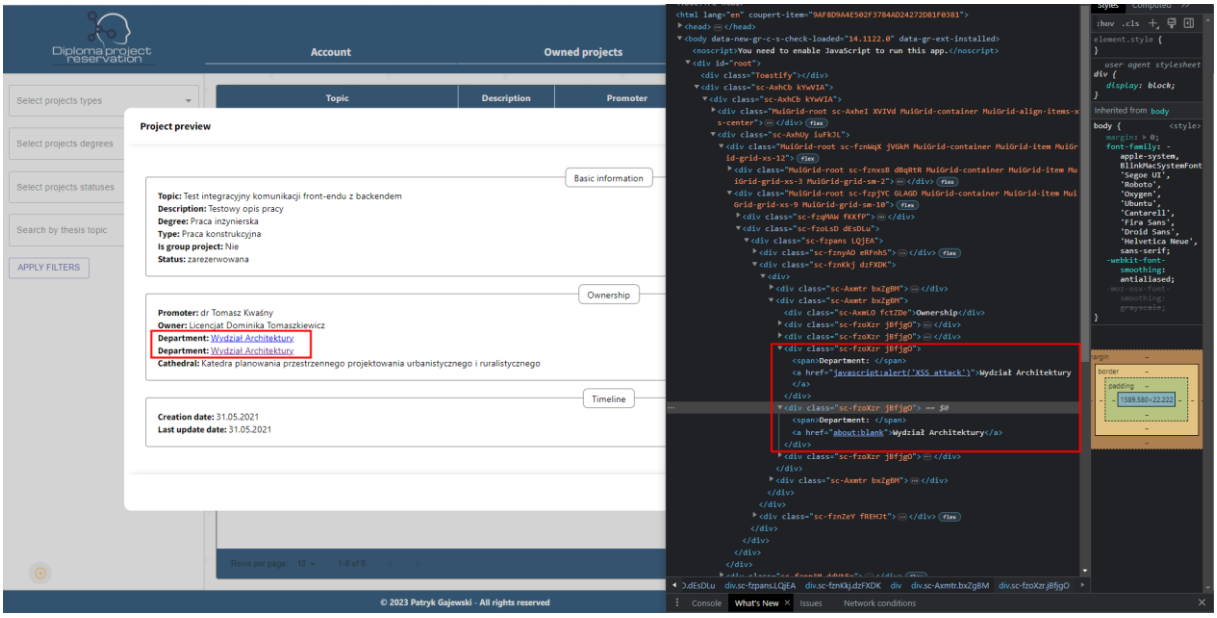
Kolejnym etapem było zaimplementowanie sanityzacji w odniesieniu do adresów hiperłączy przekazywanych do części frontendowej. W tym celu wykorzystana została biblioteka *@braintree/sanitize-url*. Do przeprowadzenia testu powielono komponent *<SectionRow/>* podając jako wartość rekwizytu *content* odpowiednio:

- bezpośrednio otrzymaną wartość adresu *URL,*
- adres *URL* przetworzony przez funkcję *sanitizeURL.*

```
<SectionRow header="Department" content={<a href={props.project.department.website}>{props.project.department.namePL.full}</a>} />
<SectionRow header="Department" content={<a href={sanitizeUrl(props.project.department.website)}>{props.project.department.namePL.full}</a>} />
<SectionRow header="Cathedral" content={props.project.cathedral.namePL} />
```

Rysunek 47 Fragment kodu wykorzystujący funkcję sanitizeURL

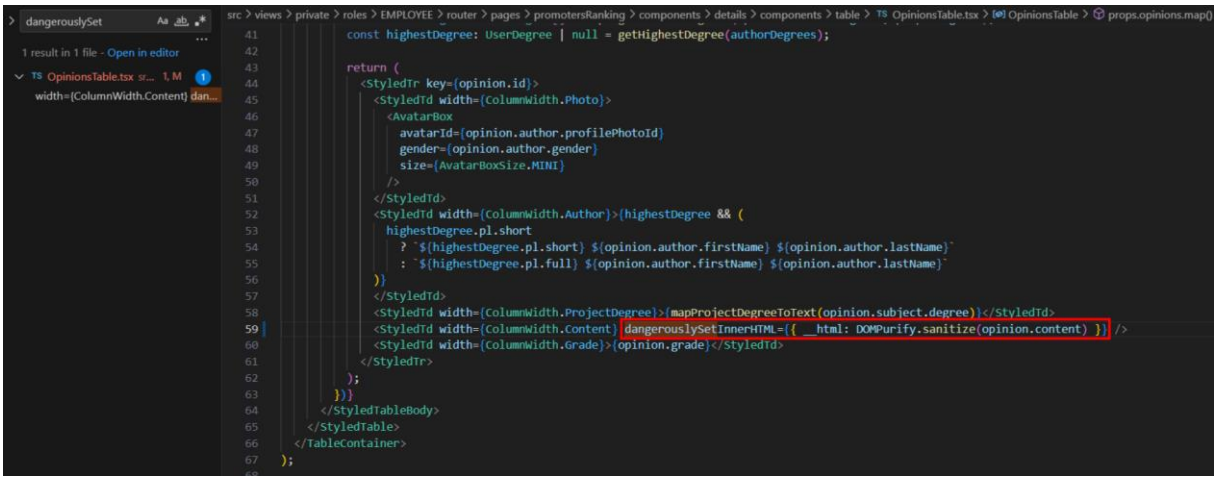
Rysunek nr 48 prezentuje efekt wykorzystania funkcji *sanitizeURL*.



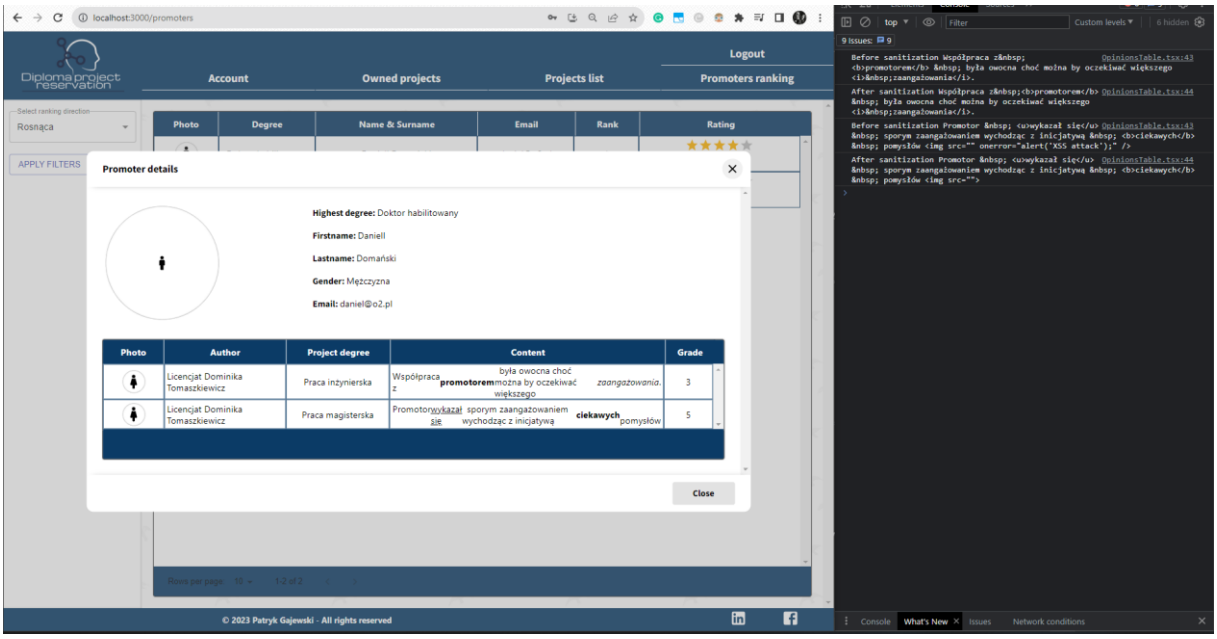
Rysunek 48 Efekt wykorzystania funkcji sanitizeURL

Efektem działania funkcji *sanitizeURL* jest zastąpienie szkodliwej zawartości atrybutu *href* wartością *about:blank*. Dzięki temu nawet po kliknięciu w hiperłącze szkodliwy kod nie zostanie wykonany. Tym samym nawet jeżeli atakującemu udałoby się zamieścić złośliwy kod w bazie danych to po stronie przeglądarki szkodliwa część zostanie unieszkodliwiona.

Ostatnim z etapów była sanityzacja danych przekazywanych do funkcji *dangerouslyInnerHTML*. Do przeprowadzenia procesu sanityzacji została wykorzystana zewnętrzna biblioteka *DOMPurify*. Rysunek nr 49 prezentuje fragment kodu, w którym użyto metody *sanitize* dostępnej w ramach biblioteki *DOMPurify*.



Rysunek 49 Sanityzacja danych z wykorzystaniem metody sanitize



Rysunek 50 Porównanie efektu przed sanityzacją i po sanityzacji

W konsoli widocznej po prawej stronie rysunku nr 50 zaprezentowano efekt wielokrotnego wywołania metody *console.log()* z zawartością komentarza. Metoda *console.log()* pozwala na wyświetlenie przekazanej wartości w konsoli przeglądarki. Pierwsze wywołanie prezentuje otrzymane dane w stanie nienaruszonym. Drugie wywołanie prezentuje jak zmieniła się zawartość komentarza po oczyszczeniu funkcją *sanitize*. W wyniku działania funkcji tag *<img>* utracił atrybut *onerror*. Dzięki temu okno informacyjne się nie pojawiło.

4. Eliminacja podatności na ataki typu Brute Force

Eliminacja podatności na ataki typu Brute Force została przeprowadzona w oparciu o bibliotekę *express-rate-limit*. Przyjęto, że po 3 nieudanych próbach logowania podjęcie kolejnej próby będzie wymagało upłynięcia określonego okresu czasu. Okres blokady został ustawiony na 10 minut.

```
const loginLimiter = rateLimit({
  windowMs: 10 * 60 * 1000, // 10 minutes
  max: 3, // Limit each IP to 3 requests per `window` (here, per 10 minutes)
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers,
  message: "To many login attempts, try later after 10 minutes"
});

authenticationRouter.post('/login',
loginLimiter,
  async (req: Request<any,any, LoginRequestBody, any>, res: Response) => {···
});
```

Rysunek 51 Fragment kodu implementujący limiter

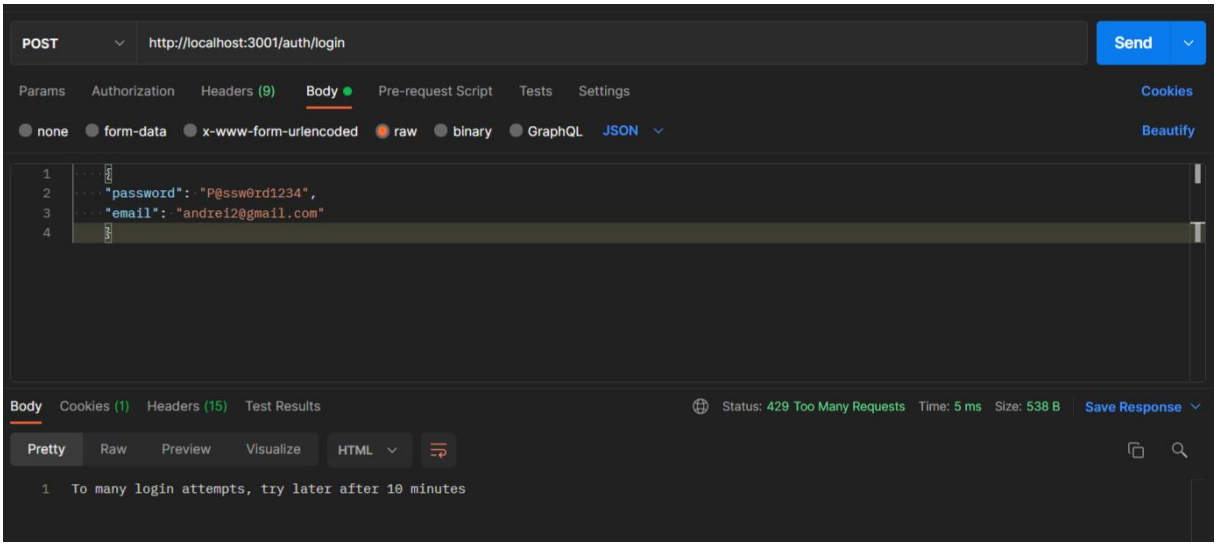
Ogranicznik zdefiniowany w ramach kodu prezentowanego na rysunku nr 51 został przekazany jako *middleware* do punktu końcowego odpowiedzialnego za logowanie użytkownika. Po wprowadzenie zmian ponownie uruchomiono test.

```
┌──(root㉿laptop)-[~]
└─# hydra -l "dominik@gmail.com" -P /home/patryk/Pulpit/passwords.txt -f 127.0.0.1 -s 3001 http-post-form "/auth/login:email=^U
SER^&password=^PASS^:S=data"
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for
illegal purposes (this is non-binding, these ***** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2023-09-04 18:20:31
[DATA] max 8 tasks per 1 server, overall 8 tasks, 8 login tries (l:1/p:8), ~1 try per task
[DATA] attacking http-post-form://127.0.0.1:3001/auth/login:email=^USER^&password=^PASS^:S=data
[ERROR] the target is using HTTP auth, not a web form, received HTTP error code 401. Use module "http-get" instead.
[ERROR] the target is using HTTP auth, not a web form, received HTTP error code 401. Use module "http-get" instead.
[ERROR] the target is using HTTP auth, not a web form, received HTTP error code 401. Use module "http-get" instead.
1 of 1 target completed, 0 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2023-09-04 18:20:32

┌──(root㉿laptop)-[~]
└─#
```

Rysunek 52 Rezultat próby złamania hasła użytkownika narzędziem hydra po wprowadzeniu blokady

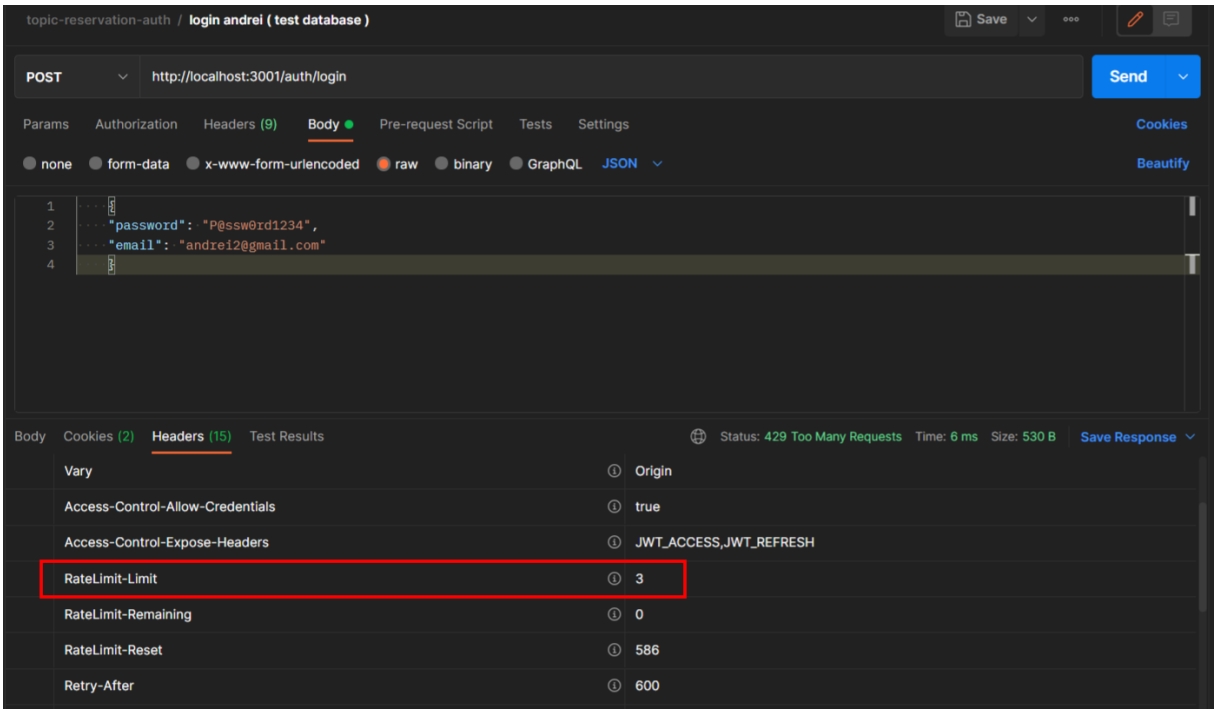
Rysunek nr 52 prezentuje przeprowadzenie wyłącznie 3 błędnych próby logowania. Potwierdza to poprawne działanie zaimplementowanego rozwiązania.

Dla pewności test został przeprowadzony również manualnie za pomocą narzędzia *Postman*. Podobnie jak w przypadku wykorzystania narzędzia *hydra* po 3 błędnych próbach logowania otrzymano następującą odpowiedź serwera.



Rysunek 53 Zapytanie i odpowiedź serwera po kilku błędnych próbach logowania

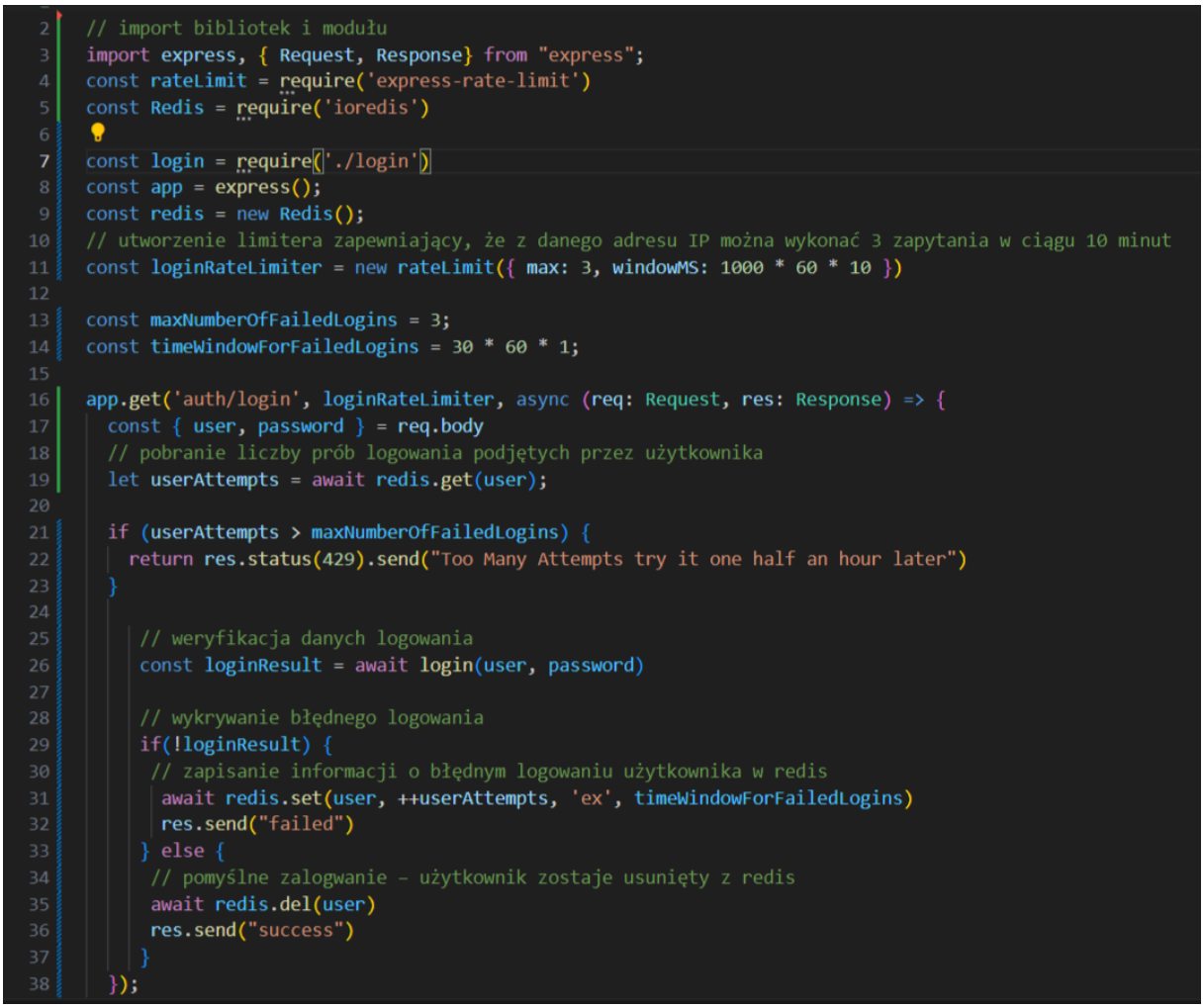
Kod błędu o numerze *429 widoczny na rysunku 53* sugeruje wykonanie zbyt dużej liczby zapytań. W ciele odpowiedzi serwera został zwrócony komunikat błędy informujący o możliwości podjęcia próby logowania po upływie 10 minut.



Rysunek 54 Sekcja nagłówkowa zapytania logowania użytkownika

Należy zwrócić uwagę na sekcję nagłówkową odpowiedzi serwera zaprezentowaną na rysunku nr 54. W tej sekcji widnieje nagłówek *RateLimit-Limit* informujący o ilości prób, które mogą zostać podjęte w określonym czasie zanim przetwarzanie kolejnych zapytań zostanie czasowo zablokowane.

W tym miejscu należy zaznaczyć, że atak może zostać przeprowadzony z wielu różnych adresów *IP* (atakujący korzysta z grupy komputerów zombie). Podatność na ataki typu Brute Force wykonywane z różnych adresów *IP* można zaimplementować w analizowanym projekcie na kilka sposobów. W projekcie wykorzystano bibliotekę *express-rate-limiter*, która została wykorzystana z domyślnymi ustawieniami. W podstawowej konfiguracji dane przechowywane były w *MemoryStore*. Obranie takiego kierunku nie pozwala na synchronizowanie zapytań. Zastosowana konfiguracja chroni przed atakami wykonywanymi z jednego adresu *IP*. W celu rozszerzenia odporności należy skorzystać np. z *Redis* będącego wysoko wydajnym systemem zarządzania bazami danych służącym do przechowywania danych w pamięci *cache*. Efekt można uzyskać na dwa sposoby. Pierwszy zakłada wykorzystanie lokalnego serwera, drugi skorzystanie z zewnętrznej usługi. W przypadku pracy wykorzystane zostało pierwsze podejście. Rysunek nr 55 prezentuje przykładową implementację limitera z wykorzystaniem *Redis*.



```
// import bibliotek i modułu
import express, { Request, Response} from "express";
const rateLimit = require('express-rate-limit')
const Redis = require('ioredis')

const login = require('./login')
const app = express();
const redis = new Redis();
// utworzenie limitera zapewniający, że z danego adresu IP można wykonać 3 zapytania w ciągu 10 minut
const loginRateLimiter = new rateLimit({ max: 3, windowMS: 1000 * 60 * 10 })

const maxNumberOfFailedLogins = 3;
const timeWindowForFailedLogins = 30 * 60 * 1;

app.get('auth/login', loginRateLimiter, async (req: Request, res: Response) => {
  const { user, password } = req.body
  // pobranie liczby prób logowania podjętych przez użytkownika
  let userAttempts = await redis.get(user);

  if (userAttempts > maxNumberOfFailedLogins) {
    return res.status(429).send("Too Many Attempts try it one half an hour later")
  }

    // weryfikacja danych logowania
    const loginResult = await login(user, password)

    // wykrywanie błędnego logowania
    if(!loginResult) {
     // zapisanie informacji o błędnym logowaniu użytkownika w redis
     await redis.set(user, ++userAttempts, 'ex', timeWindowForFailedLogins)
     res.send("failed")
    } else {
     // pomyślne zalogwanie - użytkownik zostaje usunięty z redis
     await redis.del(user)
     res.send("success")
    }
});
```

Rysunek 55 Fragment zmodyfikowanego punktu końcowego odpowiedzialnego za logowanie użytkownika

We fragmencie kodu prezentowanym na rysunku nr 55 w pierwszej kolejności zaimportowano zewnętrzne zasoby. Utworzono aplikację a także nowy obiekt *Redis*. Następnie utworzony został obiekt *loginRateLimiter*, w którym zdefiniowano limit 3 zapytań w ciągu 10 minut. W przypadku wywołania zapytania logowania na początku zostanie pobrana liczba prób logowania podjętych przez użytkownika. Następnym krokiem będzie sprawdzenie czy liczba podjętych prób nie przekroczyła założonej ilości maksymalnej. Jeżeli liczba podjętych prób przekroczyła limit to zostanie zwrócony stosowny błąd. W przeciwnym wypadku nastąpi weryfikacja poprawności danych logowania. Jeżeli podano błędne dane to ilość nieudanych prób zostanie zwiększona. W przypadku kiedy podane dane są prawidłowe obiekt użytkownika zostanie usunięty z *redis*. System działał w środowisku lokalnym co uniemożliwiło przetestowanie odporności na ataki z wykorzystaniem wielu różnych adresów *IP*. W przypadku wdrożenia systemu należy przeprowadzić takie testy aby uzyskać pewność odporności systemu na tego typu ataki.

5. Eliminacja zagrożeń atakiem Cross Site Request Forgery

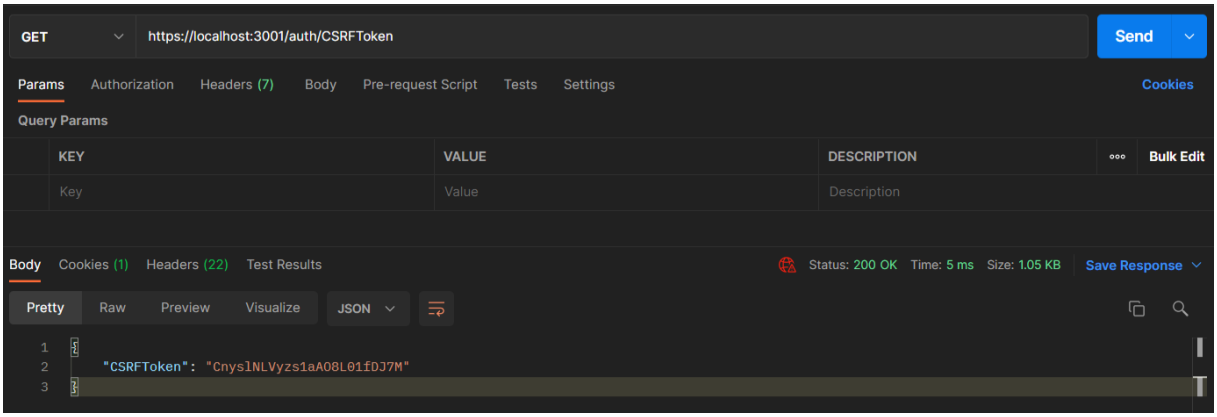
W celu zabezpieczenia przed atakami typu *Cross Site Request Forgery* wykorzystano podejście zalecane przez OWASP. Strategia zakłada wykorzystanie ciągów znaków generowanych przez serwer wyróżniających się krótkim czasem ważności. Proces implementacji logiki oparto na dedykowanej bibliotece *csrf*. Wybrana biblioteka pozwala

wygenerować *tokeny* po stronie backendu, które następnie zostaną przesłane do części frontendowej. Po stronie frontendu *tokeny* dołączone zostaną do formularzy w postaci ukrytych pól. Po zarejestrowaniu żądania zostanie zainicjowany proces sprawdzenia poprawności i aktualności wygenerowanego *tokenu*.

```
import Tokens from 'csrf';
import { authenticateMiddleware } from "middlewares/auth";

const tokens = new Tokens();

authenticationRouter.get('/CSRFToken', authenticateMiddleware,
 (req: Request, res: Response) => {
  return res.status(200).json({
    CSRFToken: tokens.create(process.env.CSRF_SECRET as string)
  });
 }
)

export const CSRFTokenMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const token = req.body.CSRFToken;
  if(tokens.verify(process.env.CSRF_SECRET as string, token)) {
    return res.sendStatus(401);
  } else {
    next();
  }
}
```

Rysunek 56 Logika generowania ciągu znaków wraz z funkcją weryfikacji poprawności

Fragment kodu widoczny na rysunku nr 56 udostępnia punkt końcowy umożliwiający pobranie utworzonego *tokenu*. Zawiera również funkcję walidacji ciągu znaków, która będzie kolejnym etapem weryfikacji następującym po autoryzacji.



Rysunek 57 Odpowiedź serwera na zapytanie o token CSRF

Rysunek nr 57 prezentuje odpowiedź backendu na zapytanie dotyczące *tokenu*.

```
useEffect(() => {
  APISecured.get('/auth/CSRFToken')
  .then((res) => {
    setCSRFToken(res.data.CSRFToken as string);
  })
  .catch((err) => {
    console.error('Cannot fetch CSRFToken')
  })
}, []);
```

Rysunek 58 Fragment kodu odpowiedzialnego za pobrane tokenu

Fragmentu kodu widoczny na rysunku nr 58 odpowiada za pobranie wartości *tokenu* przy zamontowaniu komponentu.

```
const handlePersonalSubmit = (values: PersonalDataFormValues) => {
  const dataShape = {
    firstName: values.firstName,
    lastName: values.lastName,
    birthDate: values.birthDate,
    gender: values.gender,
    degrees: values.degrees,
    address: {
      country: values.country,
      city: values.city,
      zip: values.zip,
      streetName: values.streetName,
      buildingNumber: values.buildingNumber,
      flatNumber: values.flatNumber,
    },
    CSRFToken: CSRFToken
  };

  _updateUserPersonalData(dataShape, stateData.user.id)
    .then((user: UserModel) => {
      dispatch({ ...new UpdateUserData(user) });
      toast.success('User personal data updated');
      setPersonalDataEditModalOpen(false);
    }).catch((err) => {
      toast.error('Error during personal data update');
    });
};
```

Rysunek 59 Logika załączająca token do danych przesyłanych w ramach formularza

Fragment kodu widoczny na rysunku nr 59 dołącza do zapytania pole *CSRFToken* zawierające aktualną wartość *tokenu*. Aktualna wartość została pobrana przy zamontowaniu komponentu. Tak dołączony *token* sprawdzany jest w ramach każdego punktu końcowego, w którym użyto *CSRFTokenMiddleware*.

Po wprowadzeniu zmian zagrożenie „Absence of Anti-CSRF Tokens” wykryte w ramach automatycznej analizy narzędziem OWASP ZAP się nie pojawia. Pozwala to stwierdzić poprawność zaimplementowanego rozwiązania.

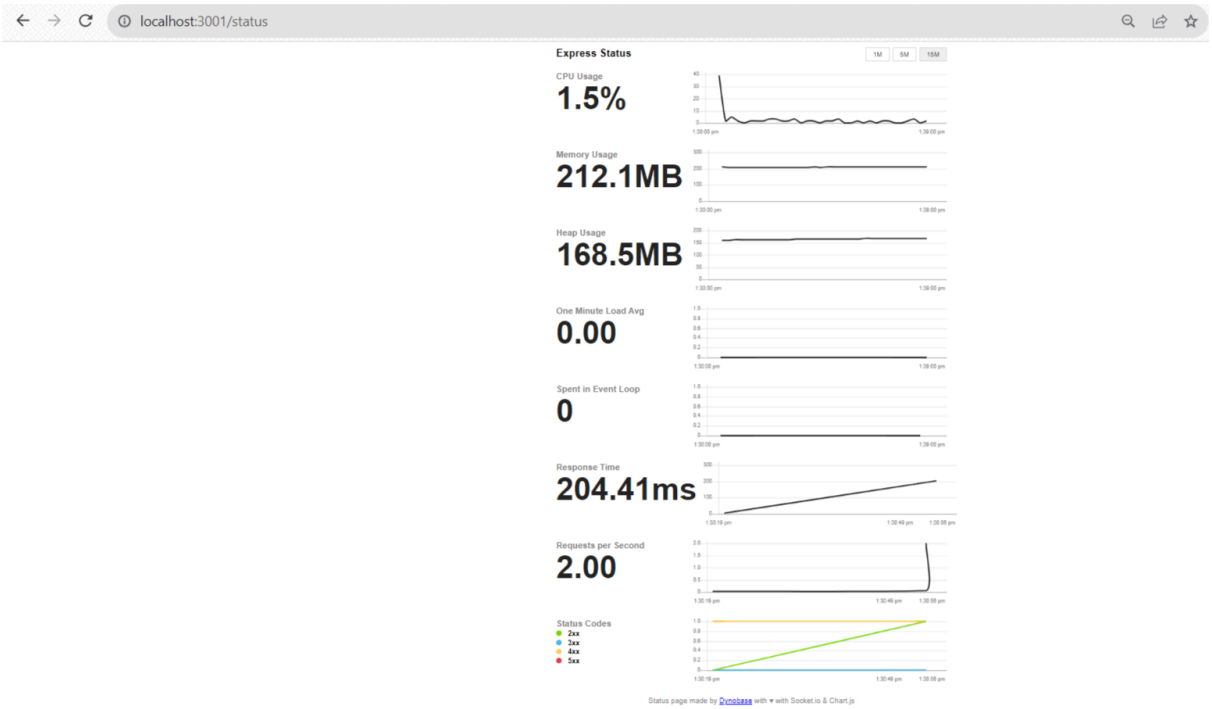
6. Wprowadzenie narzędzia do monitorowania

W celu zaimplementowania funkcjonalności monitorowania pracy backendowej części aplikacji wykorzystana została zewnętrzna biblioteka express-status-monitor. Wybrane rozwiązanie oferuje podstawową funkcjonalność monitorowania parametrów takich jak: użycie procesora, użycie pamięci, czas odpowiedzi czy wykres prezentujący zapytania podzielone w ramach kodów odpowiedzi.

```
import ESM from 'express-status-monitor';
import express from 'express';

// OUR Express instance
const app = express();

// NOTE below we ensure that all request body will be parsed as Json
app.use(bodyParser.json());

// NOTE below we ensure cookieParser
app.use(cookieParser());

app.use(ESM());
```

Rysunek 60 Użycie biblioteki express-status-monitor

Kod widoczny na rysunku nr 60 prezentuje użycie biblioteki *express-status-monitor*. Wykorzystanie biblioteki utworzyło nowy punkt końcowy pozwalający sprawdzić podstawowe parametry pracy serwera.



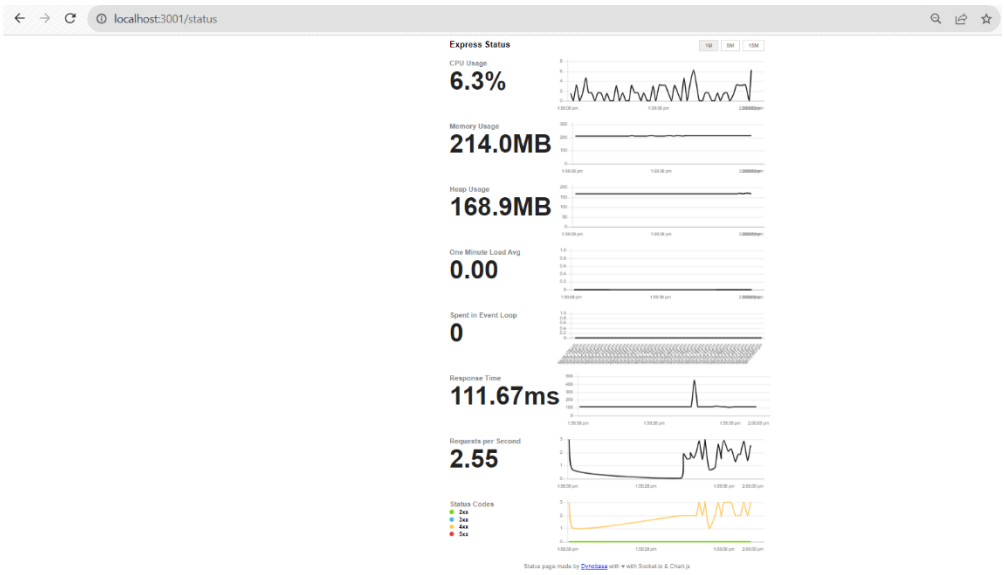
Rysunek 61 Widok monitorowania serwera

Parametry serwera widoczne na rysunku nr 61 takie jak: zużycie procesora na poziomie 1,5%, pamięci 212,5MB, czas odpowiedzi na poziomie 204.41 ms są jak najbardziej akceptowalne. W kontekście akceptowalnych wartości należy wspomnieć o wartościach, które powinny wzbudzić podejrzenia i wymagać będą weryfikacji.

Wartości wymagające dalszej weryfikacji:

- w przypadku procesora oscylującą wokół 100%,
- w przypadku pamięci oscylują wokół maksymalnej wartości dostępnej pamięci,
- w przypadku czasu odpowiedzi są na poziomie kilku sekund.

Największy wpływ na czas odpowiedzi ma poziom skomplikowania danego zapytania oraz ilość zapytań, które muszą być zrealizowane wcześniej. Poziom skomplikowania należy rozumieć jako ilość operacji do wykonania z uwzględnieniem złożoności obliczeniowej poszczególnych etapów.



Rysunek 62 Widok monitorowania serwera w momencie zwiększonego obciążenia

Rysunek nr 62 pokazuje zwiększenie ilości zapytań na sekundę. W tym wypadku wzrosło procentowe wykorzystanie mocy obliczeniowej procesora oraz ilość wykorzystywanej pamięci. Zauważyć należy spadek czasu odpowiedzi co związane jest z niskim poziomem skomplikowania zapytań, które w tym okresie były kierowane do serwera. Mimo wzrostów zużycia zasobów serwer ciągle posiada spory zapas co jest stanem oczekiwanym.

7. Eliminacja podatności zewnętrznych zależności

W celu wyeliminowania podatności zewnętrznych zależności wykorzystano komendę *npm audit fix*. Komenda udostępniona została w ramach *NPM CLI*. Polecenie pozwala rozwiązać większości problemów poprzez aktualizację zależności do nowszych wersji. Wywołanie polecenia w repozytorium backendu spowodowało następujący rezultat.

```
8 vulnerabilities (4 moderate, 4 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.
PS C:\Users\patry\Desktop\reservation_topic_project\topic-reservation-backend>
```

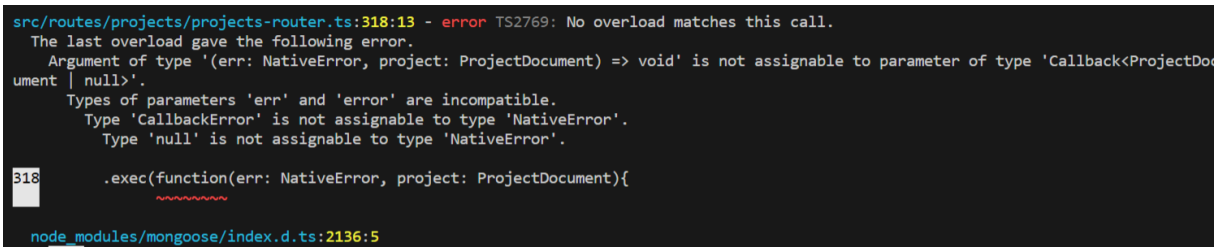
Rysunek 63 Rezultat wywołania polecenia npm audit fix

Widoczny na rysunku nr 63 rezultat pozwala odnotować spadek ilości błędów. Należy zwrócić uwagę na eliminację wszystkich błędów zakwalifikowanych do poziomu krytycznego. Wywołanie polecenia nie spowodowało usunięcia wszystkich błędów. Jest to związane ze specyfiką działania polecenia *npm audit fix*. Polecenie pozwala na rozwiązanie problemów, które zostały naprawione przez autorów poprzez udostępnienie nowej wersji zależności w odniesieniu do części ***minor*** oraz ***patch***. Dzięki temu projekt powinien uruchomić się bez większych problemów. W podstawowej formie polecenie nie będzie aktualizować zależności na poziomie **major** (zmiana na tym poziomie oznacza utratę kompatybilności). W praktyce wyeliminowanie wszystkich znanych na daną chwilę podatności często wiąże się z podniesieniem wersji na poziomie ***major***. W takim wypadku należy wykorzystać polecenie *npm audit fix* z dodatkową flagą *-force*. Po wywołaniu polecenia z flagą *-force* istnieje nikła szansa, że projekt zostanie uruchomiony poprawnie bez odpowiednich zmian w kodzie. Konieczność wprowadzenia zmian wynika z różnic w wersjach, które nie są wstecznie kompatybilne. W przypadku backendu analizowanego systemu wykorzystanie polecenia z flagą *-force* pozwoliło zmniejszyć liczbę podatności zaledwie o 1. Efekt wykonania polecenia z flagą widoczny jest na rysunku nr 64.

```
xlsx  *
Severity: high
Prototype Pollution in sheetJS - https://github.com/advisories/GHSA-4r6h-8v6p-xvw6
No fix available
node_modules/xlsx

7 vulnerabilities (3 moderate, 4 high)

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

Some issues need review, and may require choosing
a different dependency.
PS C:\Users\patry\Desktop\reservation_topic_project\topic-reservation-backend>
```

Rysunek 64 Rezultat wywołania polecenia npm audit fix -force

W przypadku pozostałych zagrożeń autorzy bibliotek nie udostępnili nowszych wersji lub nie rozwiązali w nich rozpoznanych problemów. Uruchomienie komendy *npm audit fix* z flagą *-force* spowodowało powstanie licznych błędów, które uniemożliwiały uruchomienie projektu bez wprowadzenia odpowiednich zmian w kodzie.



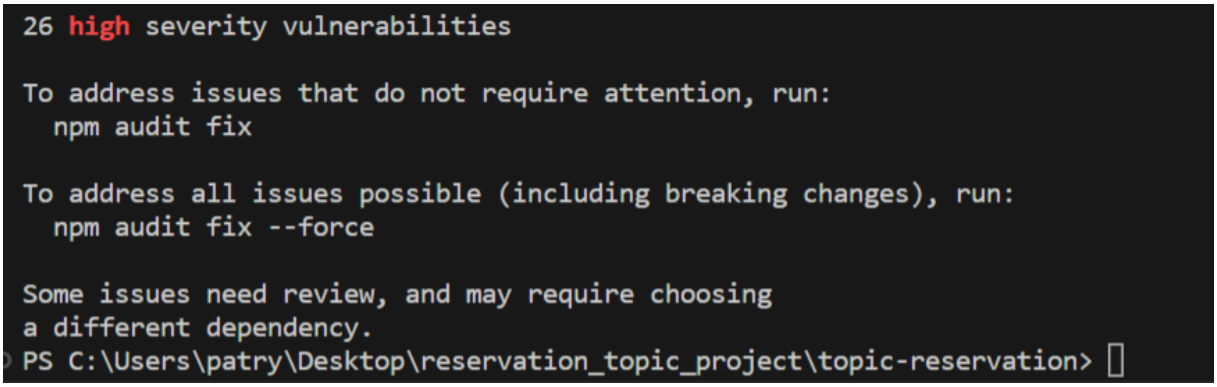
Rysunek 65 Wybrany błąd w rezultacie wywołania npm audit fix -force

Z uwagi na powstałe błędy w wyniku wywołania polecenia z flagą *-force* oraz wciąż nierozwiązane problemy wycofano zmiany wprowadzone przez *npm audit fix -force*. Następnie ponownie wywołano polecenia w podstawowym wariancie. Obranie takiego podejścia pozwoliło wyeliminować najbardziej istotne podatności bez konieczności wprowadzania rozległych zmian w kodzie.

W odniesieniu do zagrożeń wykrytych po stronie frontendu wykorzystano podobne podejście. Uruchomienie polecenia *npm audit fix* spowodowało zmniejszenie liczby podatności do 101 co zostało zaprezentowane na rysunku nr 66.

Rysunek 66 Efekty wywołania komendy npm audit fix w repozytorium części frontendowej

Ze względu na ilość pozostałych błędów widocznych na rysunku nr 66 w tym również błędów zaliczonych do poziomu krytycznego podjęto decyzję o dodatkowych krokach mających na celu zmniejszenie ilości występujących podatności. W tym celu wykonano komendę *npm audit fix -force* w wyniku czego ilość podatności zmniejszyła się drastycznie co zostało zaprezentowane na rysunku nr 67.



Rysunek 67 Efekt wywołania komendy npm audit fix -force w repozytorium frontend

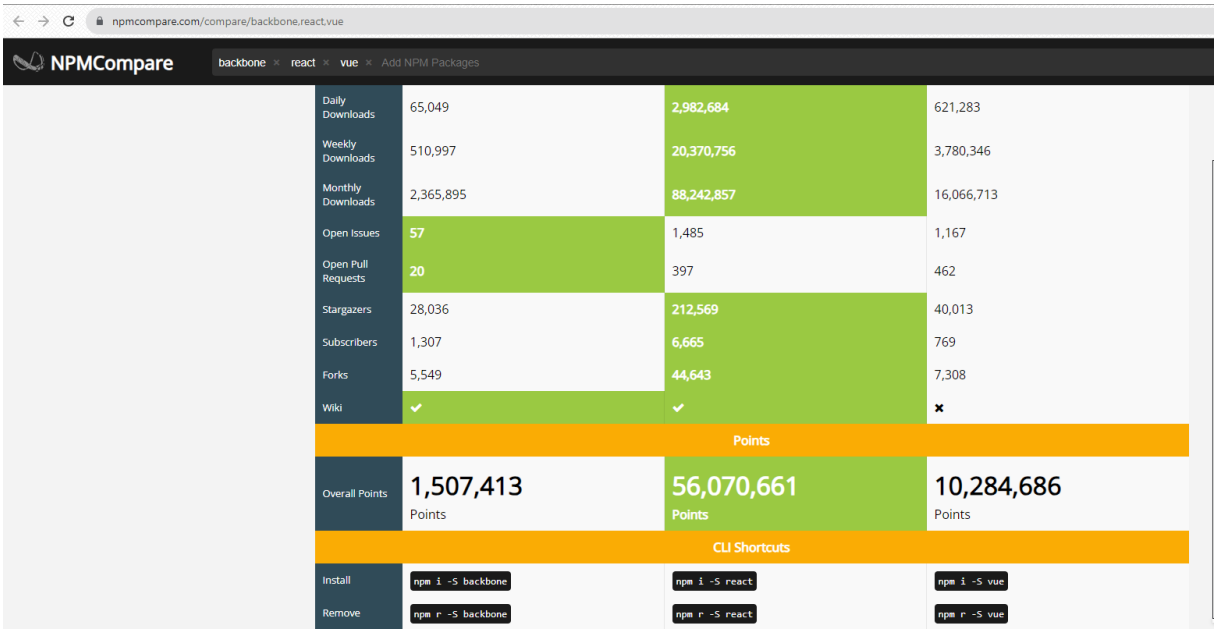
W związku z podniesieniem wersji na poziomie major wymagane były liczne zmiany w kodzie. Dopiero po ich wprowadzeniu aplikacja została prawidłowo uruchomiona.

Wnioski

Przeprowadzenie częściowej analizy poziomu bezpieczeństwa oferowanego przez autorski system pozwoliło wykryć wiele zagrożeń w badanych obszarach. Wykorzystane w tym celu narzędzia takie jak *Postman*, *OWASP ZAP* czy *npm audit* ułatwiły wykrywanie zagrożeń i prowadzenie badań. Dostęp do kodu źródłowego pozwoli testować rozwiązania redukujące i eliminujące wykryte w ramach analizy podatności. Wyeliminowanie pewnych podatności wymagało rozległych zmian w kodzie źródłowym jak to miało miejsce w przypadku eliminacji podatności zewnętrznych zależności. W przypadku pozostałych podatności modyfikacji zostały poddane wyłącznie niewielkie fragmenty kodu źródłowego. W wielu przypadkach wykorzystano zewnętrzne biblioteki oferujące poszukiwaną funkcjonalność. Prawie wszystkie podatności wykryte w ramach analizy zostały wyeliminowane co jest efektem zadowalającym. Przeprowadzenie analizy wiązało się z koniecznością pozyskania wiedzy zarówno w wymiarze praktycznym jak i teoretycznym. W związku z częściowym charakterem analizy należy zasugerować wykonanie dalszych działań, które z uwagi na obszerność tematu zostały zaniechane. Analiza pozostałych domen będzie szczególnie interesująca w przypadku kiedy rozwiązanie zostanie wdrożone produkcyjnie. Otworzyłoby to wykonania pełnego audytu bezpieczeństwa.

Perspektywy

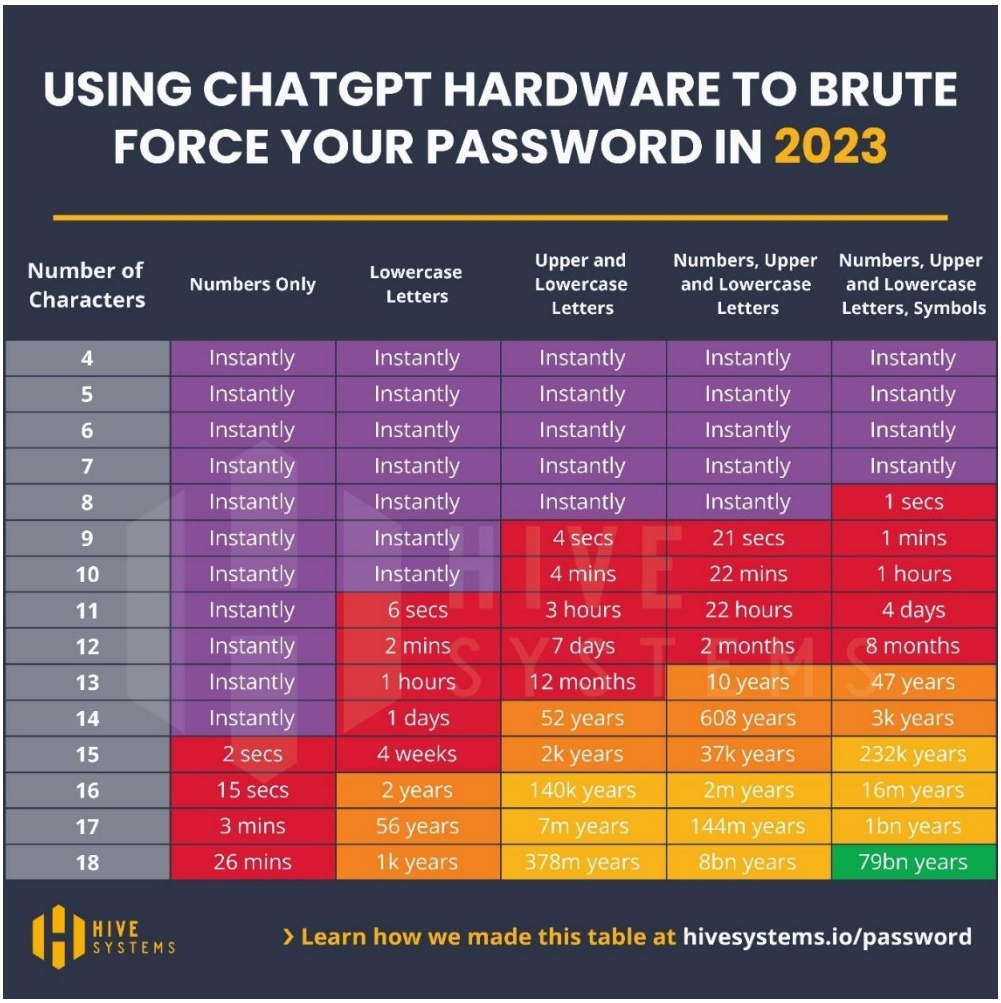
Stawiając bezpieczeństwo na pierwszym miejscu należałoby rozwiązać problem pozostałych podatności występujących w zewnętrznych bibliotekach. Na tym etapie warto upewnić się czy aby na pewno fragmenty biblioteki zawierające podatność są wykorzystywane w projekcie. Może się okazać, że podatne części bibliotek nie są wykorzystywane. W takim przypadku daną podatność można pominąć. Jeżeli podatności dotyczą wykorzystywanych w projekcie części bibliotek należy poszukać rozwiązań alternatywnych. W przypadku wielu dostępnych alternatyw w wyborze pomocne może okazać się narzędzie *NPMCompare*. Narzędzie dostępne jest pod adresem: https://npmcompare.com/. Pozwala porównać biblioteki pod względem takich wyznaczników jak m. in.: liczba zależności, rodzaj licencji czy ilość osób zaangażowanych w rozwój.



Rysunek 68 Zestawienie bibliotek: Backbone, React i Vue w NPMCompare

W przykładnie zaprezentowanym na rysunku nr 68 porównano biblioteki *Backbone*, *React* i *Vue*. Na podstawie liczby uzyskanych punktów za poszczególne wyznaczniki najwyżej sklasyfikowana została biblioteka *React*. Wynik analizy przeprowadzonej za pomocą narzędzie *NPMCompare* powinien mieć wpływ na wybór ostatecznego rozwiązania. W przypadku braku alternatyw należy rozważyć stworzenie własnych rozwiązań jeżeli posiadamy odpowiednie zasoby.

W odniesieniu do kwestii rejestracji warto rozważyć wprowadzenie bardziej rygorystycznej polityki haseł, których minimalna długość będzie wynosić np. 10 znaków. Podjęcie takiej decyzji może mieć negatywny wpływ na wygodę użytkowania rozwiązania. Wprowadzenie wymagania bardziej złożonego hasła pozwoliłoby znacząco wydłużyć czas potrzebny na złamanie hasła metodą siłową. Dzięki temu przy zachowaniu pozostałych zasad złożoności hasła potrzebna byłaby przynajmniej godzina na złamanie takiego hasła (w porównaniu do tylko 1s przy 8 znakowym trudnym haśle). Przedstawione kalkulacje dotyczącą stanu na 2023 przy wykorzystaniu technologii zaawansowanych modeli sztucznej inteligencji i dotyczą łamania haseł metodą siłową. Rysunek nr 69 przestawia tabelę prezentującą zależność długość czasu potrzebnego na złamanie hasła metodą Brute Force z wykorzystaniem modelu sztucznej inteligencji w zależności od długości i poziomu złożoności hasła.



Rysunek 69 Tabela zależności długości czasu potrzebnego na złamanie hasła metodą siłową z wykorzystaniem modeli uczenia maszynowego w zależności od złożoności

W przypadku wdrożenia rozwiązania w odniesieniu do polityki nagłówków odpowiedzi serwera logicznym rozwiązaniem będzie przeprowadzenie procesu tworzenia polityki na kopii środowiska produkcyjnego co pozwoli uniknąć olbrzymiej ilości problemów z dostępnością. Problemy dotyczące ograniczenia znaków wprowadzonych w ramach polityki można rozwiązać poprzez ograniczanie zapisów z postaci subdomen do domen, względem których mamy największe zaufanie. Dzięki temu długość zapisów istotnie spadnie. Dodatkowo warto upewnić się czy dany zasób już wcześniej nie występuje i czy można go bardziej uogólnić.

W przypadku wdrożenia produkcyjnego rozwiązania sugerowane jest przeprowadzenie szeregu testów powdrożeniowych, które mogą ujawnić pozostałe luki w niezbadanych w ramach pracy obszarach. Nie bez znaczenia jest również aspekt raportowania błędów, który został pominięty w odniesieniu do części frontendowej. Kwestia podstawowego monitorowania została zaimplementowana wyłącznie w odniesieniu do backendu. Wybrane rozwiązanie nie dostarcza jednak funkcjonalności raportowania. W tym przypadku użycia produkcyjnego należy rozważyć użycie bardziej skomplikowanych narzędzia jak np. *AppSignal* czy *PM2* do monitorowania części backendowej aplikacji. W przypadku narzędzi do monitorowania i raportowania po stronie frontendu można rozważyć wykorzystanie rozwiązań *Sentry* czy *Datadog*.

Podsumowanie

Podczas przeprowadzonej częściowej analizy ujawniono szereg nieprawidłowości w stworzonym uprzednio systemie. Taki stan rzeczy częściowo wytłumaczyć można skupieniem autora na dostarczeniu rozbudowanego, działającego *PoC (ang. Proof of Concept)* rozwiązania. Podobne sytuacje nie są odosobnione choć zdecydowanie częściej elementy bezpieczeństwa systemu testowane są na wcześniejszych etapach cyklu życia oprogramowania. W przypadku kiedy kwestie bezpieczeństwa testowane są wyłącznie na ostatnim etapie należy przeprowadzić kompleksową analizę rozwiązania. Analiza powinna być zorientowana na wykryciu obszarów, w których należy podjąć odpowiednie działania na rzecz zwiększenia bezpieczeństwa. Wybrane obszary bezpieczeństwa testowane w ramach pracy mogą okazać się pomocne. Celem wykonania kompletnej analizy bezpieczeństwa systemu należy zwrócić uwagę również na pozostałe aspekty. W ramach pracy zaprezentowano konkretne rozwiązania pozwalające zredukować bądź wyeliminować podatności w badanych obszarach. Należy zaznaczyć, że podjęte strategie i użyte narzędzia są jedynie propozycją rozwiązania danego problemu. Obrane strategie zostały dostosowane do testowanego rozwiązania. W przypadku innych projektów warto rozważyć wykorzystanie bardziej adekwatnych strategii i dedykowanych bibliotek. W ramach pracy osiągnięto założone cele. Proces weryfikacji wybranych obszarów pozwolił wychwycić luki, które w wyniku podjęcia konkretnych działań zostały zminimalizowane lub wyeliminowane. Przeprowadzenie analizy oraz wprowadzenie stosownych zmian pozwoliło zwiększyć poziom bezpieczeństwa analizowanego systemu. Realizacja tematu pracy pozwoliła zwiększyć poziom wiedzy na temat testowania bezpieczeństwa jak i również stosowania technik i narzędzi zwiększających bezpieczeństwo. Wiedza pozyskana w ramach realizacji pracy pozwoli tworzyć bardziej bezpieczne rozwiązania w przyszłości. Mimo analizy wyłącznie wybranych fragmentów mam nadzieję, że praca pozwoliła zobrazować jak skomplikowany i wieloaspektowy jest proces wykrywania, przeciwdziałania i usuwania podatności aplikacji internetowych.

Spis rysunków

Spis tabel

Bibliografia

[1] 1-02, J. P. (2010, 11 8). Department of Defense Dictionary of Military and Associated Terms.

[2] Andress, J. (2021). *Podstawy bezpieczeństwa informacji. Praktyczne wprowadzenie.* Helion.

[3] Centrum Badania Opinii Publicznej. (2022, Czerwiec). *Korzystanie z internetu w 2022 roku.* Pobrano Wrzesień 1, 2022 z lokalizacji https://www.cbos.pl/SPISKOM.POL/2022/K_077_22.PDF

[4] CYBERARK. (2023). *2023 Identity Security Threat Landscape Report.*

[5] Designveloper. (2022, Czerwiec 18). *Web Application Security: The Best Guide and Its Practicies*. Pobrano z lokalizacji https://www.designveloper.com/blog/web-application-security/

[6] Dyjak, A. (2021, Sierpień 19). *Bezpieczny Kod*. Pobrano z lokalizacji https://blog.bezpiecznykod.pl/owasp-wstg-mstg-opc-ocss/

[7] Fałdowski, M. (2018, 2). Zeszyty Naukowe SGSP. *Współczesny wymiar bezpieczeństwa, 2(66)*.

[8] Gilberto Najera-Gutierrez, J. A. (2019). *Kali Linux Testy penetracyjne* (wyd. 3). (G. Kowalczyk, Tłum.) Helion.

[9] Halachev, P. (2019). Cross-site scripting attacks and the security of web applications. *International Scientific Journal Security & Future*.

[10] Jerzy, S. (1996). *Współczesne pojmowanie bezpieczeństwa.* Warszawa.

[11] Konstytucja Rzeczypospolitej Polskiej, Dzienniku Ustaw: Dz.U. z 1997 r. nr 78, poz. 483.

[12] Korzeniowski, L. (2008). *Securitologia. Nauka o bezpieczeństwie człowieka i organizacji społecznych.* Kraków.

[13] Korzeniowski, L. (2010). *Menedżement.* Kraków: EAS.

[14] Koziej, S. (2011). Bezpieczeństwo: istota, podstawowe kategorie i historyczna ewolucja. *Polityczno-strategiczne aspekty bezpieczeństwa, II*.

[15] Koziej, S. (2015, Maj 5). *Wstęp do teorii i historii bezpieczeństwa.* Pobrano z lokalizacji https://koziej.pl/wp-content/uploads/2015/05/Teoria_i_historia_bezpieczenstwa.doc

[16] Kruk, T. (2021). Praktyczne modelowanie zagrożeń dla systemów teleinformatycznych z wykorzystaniem modelu STRIDE. *Pomiary Automatyka Robotyka*(4).

[17] Liderman, K. (2002). Standardy w ocenie bezpieczeństwa teleinformatycznego. *Biuletyn Instytutu Automatyki i Robotyki WAT*(17).

[18] Liedel, K. (2008). *Bezpieczeństwo informacyjne w dobie terrorystycznych i innych zagrożeń bezpieczeństwa narodowego.* Toruń: Wydawnictwo Adam Marszałek.

[19] Madej, J. (2010). Klasyfikacja zagrożeń bzpieczeństwa systemu informatycznego. *Zeszyty Naukowe*(814).

[20] Marek Antczak, Z. Ś. (2013). Metodyki testowania bezpieczeństwa aplikacji internetowych. *Przegląd teleinformatyczny*(2), strony 13-38.

[21] Maurice Dawson, D. N. (2010). Integrating Software Assurance into the Software Development Life Cycle (SDLC). *Journal of Information Systems Technology and Planning, 3*, strony 49-53.

[22] Morgan, S. (2022). *2022 Official Cybercrime Report.* Cybersecurity Ventures.

[23] Nader, K. (2023, Marzec 18). *AppMaster*. Pobrano Sierpień 15, 2023 z lokalizacji https://appmaster.io/pl/blog/metodologia-waterfall

[24] Narodowy Standard Cyberbezpieczeństwa. (2021). Słownik kluczowych pojęć z zakresu cyberbezpieczeństwa. Warszawa.

[25] NPM. (2016, Styczeń 8). *npmjs.com*. Pobrano Wrzesień 2, 2023 z lokalizacji https://docs.npmjs.com/about-audit-reports

[26] OWASP. (2020). *Web Security Tesing Guide.* Pobrano Sierpień 22, 2023 z lokalizacji https://owasp.org/www-project-web-security-testing-guide/v42/

[27] Remigiusz, R. (2010). O pojęciu i istocie bezpieczeństwa. Poznań.

[28] Rodzaje cyberzagrożeń - zagrożenia nietechniczne (społeczne). (2023, Sierpień 20). Pobrano z lokalizacji https://www.gov.pl/web/baza-wiedzy/zagroz-nietechniczne-spoleczne

[29] Siczek, A. (2019, Listopad 28). Pobrano z lokalizacji Co to jest i jak dziala JSON Web Tokens: https://global4net.com/blog/ecommerce/co-to-jest-i-jak-dziala-json-web-tokens-jwt/

[30] Simson Garfinkel, G. S. (2003). *Practical Unix and Internet Security.*

[31] Siwicki, M. (2013). *Cyberprzestępczość.* Warszawa.

[32] Stańczyk, J. (2017). *Formułowanie kategorii pojęciowej bezpieczeństwa.* Fundajca na rzecz Czystej Energii.

[33] Taylor, P. (2023). *Global internet user growth 2018-2023.* Pobrano z lokalizacji https://www.statista.com/statistics/1190263/internet-users-worldwide/

[34] Titus Winters, T. M. (2023). *Inżynieria oprogramowania według Google. Czego warto się nauczyć o tworzeniu oprogramowania.* Helion.

[35] Uniwersytet Pedagogiczny im. Komisji Edukacji Narodowej. (2019). *Vademecum Bezpieczeństwa Informacyjnego* (Tom 1). (R. K. Olga Wasiuta, Red.) Kraków.

[36] Wasilewski, J. (2017). Cyberprzestępczość - wybrane aspekty prawokarne i kryminalistyczne. *Praca doktorska napisana w Katedrze Prawa Karnego pod kierunkiem dr. hab Ewy M. Guzik-Makaruk*. Białystok.

[37] Wilson, J. (2013). *Node.js the RIght Way Practical, Server-Side JavaScript That Scales.* Dallas: The Pragmatic Programmers.

[38] Wolski, M. (2019, Grudzień 18). Modelowanie zagrożeń - teoretycznie. Pobrano z lokalizacji https://wolski.pro/2019/12/modelowanie-zagrozen-teoretycznie/

[39] *Wprowadzenie do REST API*. (2023, Kwiecień 30). Pobrano Sierpień 22, 2023 z lokalizacji https://devszczepaniak.pl/wprowadzenie-do-rest-api/

[40] Wróbel, M. (2014). *Cyberprzestępczość w Polskim systemi prawnym.* Pobrano Sierpień 25, 2023 z lokalizacji http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-3b64861c-33fb-4616-a700-59f3db1969e4

[41] Zięba, R. (2012). O tożsamości nauk o bezpieczeństwie. *Zeszyty Naukowe Akademii Obrony Narodowej*(1(86)), 7-22.