Obraz zawierający Grafika, symbol, logo, Jaskrawoniebieski

Opis wygenerowany automatycznie

**UNIWERSYTET RZESZOWSKI**

**Wydział Nauk Ścisłych i Technicznych**

Maciej Biel

117780

Informatyka

**Bezpieczeństwo aplikacjiweb w Ruby on Rails**

Praca magisterska

Praca wykonana pod kierunkiem

Dr inż. Marcin Ochab

Rzeszów, 2025

Spis treści

[1 Wprowadzenie 3](#_Toc194839636)

[1.1 Problematyka pracy 4](#_Toc194839637)

[1.2 Cel i zakres pracy 5](#_Toc194839638)

[1.3 Uzasadnienie innowacyjności tematyki 6](#_Toc194839639)

[2 Bezpieczeństwo aplikacji webowych 7](#_Toc194839640)

[2.1 OWASP – misja organizacji i wpływ na standardy bezpieczeństwa 8](#_Toc194839641)

[2.2 Przegląd dokumentu OWASP Top 10 (2021) 9](#_Toc194839642)

[3 Ruby on Rails 11](#_Toc194839643)

[3.1 Koncepcje architektoniczne i podejście do tworzenia aplikacji 11](#_Toc194839644)

[3.2 Ruby on Rails jako stabilne środowisko tworzenia i rozwijania aplikacji webowych 11](#_Toc194839645)

[4 Struktura repozytorium z przykładami podatności 12](#_Toc194839646)

[4.1 Organizacja monorepozytorium z przykładami 14](#_Toc194839647)

[4.2 Uruchamianie aplikacji testowych z wykorzystaniem Docker Compose 15](#_Toc194839648)

[5 Analiza wybranych podatności z dokumentu OWASP Top 10 (2021) w kontekście Ruby on Rails 16](#_Toc194839649)

[5.1 A01: Broken Access Control 16](#_Toc194839650)

[5.1.1 Mass Assignment 16](#_Toc194839651)

[5.1.2 Insecure Direct Object References (IDOR) 20](#_Toc194839652)

[5.1.3 Local File Inclusion (LFI) 26](#_Toc194839653)

[5.1.4 Object Level Authorization (OLA) 26](#_Toc194839654)

[5.2 A02:2021 - Cryptographic Failures 26](#_Toc194839655)

[5.2.1 Weak Encoding for Password 26](#_Toc194839656)

[5.2.2 Use of Hard-Coded Cryptographic Key 26](#_Toc194839657)

[5.2.3 Cleartext Transmission of Sensitive Information 26](#_Toc194839658)

[5.3 A03:2021 – Injection 27](#_Toc194839659)

[5.3.1 Remote Code Execution (RCE) 27](#_Toc194839660)

[5.3.2 Cross-Site Scripting (XSS) 27](#_Toc194839661)

[5.3.3 Cross-Site Request Forgery (CSRF) 27](#_Toc194839662)

[5.3.4 SQL Injection (SQLi) 27](#_Toc194839663)

[5.4 A04:2021 - Insecure Design 27](#_Toc194839664)

[5.4.1 Open Redirect 27](#_Toc194839665)

[5.4.2 Regex Denial of Service (ReDoS) 28](#_Toc194839666)

[5.4.3 Login Rate Limiting 28](#_Toc194839667)

[5.5 A05:2021 - Security Misconfiguration 28](#_Toc194839668)

[5.5.1 Token / Cookie Misconfiguration 28](#_Toc194839669)

[5.5.2 XML XXE 28](#_Toc194839670)

[5.5.3 TLS Force / HSTS 28](#_Toc194839671)

[5.5.4 Debug Mode 29](#_Toc194839672)

[5.6 A09:2021 - Security Logging and Monitoring Failures 29](#_Toc194839673)

[5.6.1 Enable Logging 29](#_Toc194839674)

[5.6.2 Logging Sensitive Information 29](#_Toc194839675)

[5.6.3 No Log to User 29](#_Toc194839676)

[5.7 A10:2021 - Server-Side Request Forgery 29](#_Toc194839677)

[5.7.1 Server-Side Request Forgery 29](#_Toc194839678)

[6 Zakończenie 30](#_Toc194839679)

[Bibliografia 31](#_Toc194839680)

[Wykaz rysunków 32](#_Toc194839681)

[Wykaz zdjęć i grafik z Internetu użytych w pracy 33](#_Toc194839682)

# Wprowadzenie

W dobie powszechnej cyfryzacji, aplikacje webowe odgrywają kluczową rolę w niemal każdym aspekcie codziennego życia – od bankowości elektronicznej, przez komunikację i zakupy online, aż po zarządzanie danymi w przedsiębiorstwach.

Z każdym rokiem rośnie liczba usług świadczonych w oparciu o technologię webową, a wraz z nią – zależność użytkowników oraz organizacji od niezawodności i bezpieczeństwa tych rozwiązań. Dynamiczny rozwój technologii informatycznych niesie za sobą wiele korzyści, ale również stwarza nowe wyzwania.

Jednym z najistotniejszych z nich jest zapewnienie bezpieczeństwa systemów, które są dostępne z poziomu przeglądarki internetowej. W odróżnieniu od tradycyjnych aplikacji, systemy webowe są narażone na szereg specyficznych zagrożeń wynikających z ich otwartego charakteru i szerokiego zakresu interakcji z użytkownikami oraz z innymi systemami.

Problem bezpieczeństwa nie ogranicza się jedynie do aspektów technicznych, takich jak kod źródłowy czy konfiguracja serwera. Coraz częściej pojawia się potrzeba zrozumienia bezpieczeństwa jako integralnego elementu procesu projektowego, uwzględnianego na każdym etapie tworzenia oprogramowania. Z tego względu rośnie znaczenie edukacji, standaryzacji i promowania dobrych praktyk, które pomagają zidentyfikować oraz wyeliminować najczęstsze błędy popełniane podczas tworzenia aplikacji webowych.

W kontekście stale rosnącej liczby incydentów związanych z naruszeniem danych oraz coraz większej odpowiedzialności prawnej i finansowej spoczywającej na dostawcach usług cyfrowych, bezpieczeństwo aplikacji staje się nie tylko kwestią techniczną, ale również biznesową i społeczną. Dlatego zagadnienia związane z identyfikacją podatności oraz ich skutecznym przeciwdziałaniem zyskują coraz większe znaczenie w środowisku programistycznym, akademickim i komercyjnym.

## Problematyka pracy

Wraz z rozwojem Internetu i technologii webowych coraz więcej procesów – zarówno tych codziennych, jak i krytycznych dla firm czy instytucji – przenosi się do aplikacji działających w przeglądarce. To wygodne, szybkie i często tanie rozwiązania. Ale za tą dostępnością kryje się poważny problem: bezpieczeństwo. A konkretniej – jego brak, wynikający z błędów, które bardzo łatwo popełnić podczas tworzenia oprogramowania.

Podatności aplikacji webowych to zjawisko powszechne. Co więcej, wiele z nich nie wynika z jakichś egzotycznych, trudnych do zrozumienia zagrożeń. Przeciwnie – są one dobrze znane, opisane, powtarzalne. A mimo to wciąż pojawiają się w nowych projektach.

Wynika to m.in. z faktu, że deweloperzy – zwłaszcza mniej doświadczeni – koncentrują się na funkcjonalności i szybkim dostarczeniu rozwiązania, a kwestie bezpieczeństwa spychane są na dalszy plan. Takie podejście może prowadzić do poważnych konsekwencji, zwłaszcza gdy aplikacja operuje na danych wrażliwych.

Frameworki webowe, takie jak Ruby on Rails, dostarczają wielu rozwiązań, które mają chronić przed typowymi atakami. Problem w tym, że nie zawsze są one używane właściwie. Czasem nieświadomość, czasem błędna konfiguracja, a czasem po prostu pośpiech sprawiają, że gotowe zabezpieczenia nie działają tak, jak powinny – albo w ogóle nie są używane. To otwiera drogę do ataków, których można by łatwo uniknąć.

Zdarza się też, że nawet znając podstawowe zasady bezpieczeństwa, programiści nie wiedzą, jak je zastosować w konkretnym środowisku. Wiedza książkowa rozjeżdża się z praktyką. Dlatego tak istotne staje się nie tylko opisywanie samych podatności, ale też pokazywanie ich działania „na żywym organizmie”, w konkretnym kontekście technologicznym. Tylko wtedy można naprawdę zrozumieć, na czym polega błąd – i jak go nie popełniać.

## Cel i zakres pracy

Celem tej pracy jest przyjrzenie się bezpieczeństwu aplikacji webowych przez pryzmat konkretnych, realnych zagrożeń – takich, które nie są jedynie teorią z podręczników, ale faktycznie występują w projektach opartych na popularnych frameworkach. W centrum zainteresowania znajduje się środowisko Ruby on Rails, a główną osią analizy są podatności zdefiniowane przez OWASP w zestawieniu Top 10.

Choć framework ten oferuje szereg wbudowanych mechanizmów chroniących przed najczęstszymi błędami, nie oznacza to, że jest całkowicie odporny. Zdarza się, że zabezpieczenia są niepełne, źle zastosowane albo zupełnie pomijane. Właśnie te przypadki są przedmiotem analizy – zarówno z perspektywy tego, jak dochodzi do błędu, jak i tego, co może się wydarzyć, gdy podatność zostanie wykorzystana.

Zakres pracy obejmuje wybrane podatności bezpieczeństwa z listy OWASP Top 10 (2021), które zostały przedstawione i przeanalizowane w kontekście frameworka Ruby on Rails. Dla każdej z nich przygotowano opis teoretyczny oraz – tam, gdzie było to zasadne – praktyczny przykład działania podatności w formie uruchamialnej aplikacji. Przykłady te zostały zorganizowane w repozytorium GitHub w postaci monorepozytorium, zawierającego strukturę folderów odpowiadających konkretnym podatnościom.

Analiza nie ma na celu wyczerpania całego tematu bezpieczeństwa aplikacji webowych, ale skupia się na najczęściej występujących zagrożeniach i ich obecności (lub braku) w środowisku Ruby on Rails. Praca ma charakter zarówno edukacyjny, jak i praktyczny – może służyć jako punkt wyjścia do dalszych badań lub jako materiał pomocniczy w procesie nauki bezpiecznego tworzenia aplikacji.

## Uzasadnienie innowacyjności tematyki

Bezpieczeństwo aplikacji webowych stanowi obecnie jeden z kluczowych obszarów w dziedzinie tworzenia oprogramowania. W miarę jak rośnie liczba usług i procesów przenoszonych do środowisk internetowych, wzrasta również powierzchnia ataku, a tym samym – ryzyko wystąpienia podatności.

Mimo istnienia licznych wytycznych, standardów oraz dostępności narzędzi wspierających bezpieczne programowanie, błędy związane z bezpieczeństwem nadal są powszechnie spotykane w nowo tworzonych aplikacjach. Zjawisko to nie wynika z braku świadomości, lecz z rosnącej złożoności środowisk technologicznych, szybko zmieniających się zależności oraz różnic w implementacjach pomiędzy frameworkami. Wiele podatności występujących w aplikacjach webowych – takich jak błędna walidacja danych, niewłaściwe zarządzanie sesją czy brak kontroli dostępu – pozostaje niezmienne od lat, jednak przybierają one różne formy w zależności od użytej technologii. To powoduje, że skuteczne przeciwdziałanie zagrożeniom wymaga nie tylko wiedzy teoretycznej, ale również znajomości konkretnego środowiska i jego specyfiki.

Jednocześnie obserwuje się systematyczny wzrost zapotrzebowania na specjalistów ds. bezpieczeństwa aplikacji. W globalnych raportach branżowych zwraca się uwagę na deficyt wykwalifikowanej kadry, zdolnej do projektowania, audytowania i rozwijania systemów informatycznych z uwzględnieniem aspektów bezpieczeństwa. Według analiz rynkowych, brak ten dotyczy nie tylko obszaru testów penetracyjnych, ale także kompetencji deweloperskich, obejmujących bezpieczne tworzenie oprogramowania. W tym kontekście podejmowanie tematów związanych z analizą podatności w odniesieniu do konkretnych technologii – takich jak Ruby on Rails – ma istotne znaczenie dydaktyczne i praktyczne.

Praca skupiająca się na tym zagadnieniu odpowiada na realne potrzeby rynku, a jednocześnie wypełnia lukę w dostępnych zasobach edukacyjnych. Zestawienie wiedzy teoretycznej z przykładami implementacyjnymi pozwala lepiej zrozumieć charakter zagrożeń oraz ułatwia nabywanie kompetencji niezbędnych do ich skutecznego eliminowania w praktyce programistycznej.

# Bezpieczeństwo aplikacji webowych

Aplikacje webowe stanowią trzon nowoczesnych systemów informatycznych. Obsługują bankowość, e-commerce, platformy edukacyjne, administrację publiczną i wiele innych obszarów życia codziennego. Ich powszechna dostępność sprawia, że są wygodne i łatwo skalowalne, ale jednocześnie podatne na wiele zagrożeń – zarówno ze strony złośliwych użytkowników, jak i błędów popełnianych na etapie ich tworzenia.

W odróżnieniu od aplikacji działających lokalnie, systemy webowe są stale narażone na kontakt z otoczeniem zewnętrznym. Dane wejściowe pochodzą od nieznanych użytkowników, a ich przepływ obejmuje liczne warstwy systemowe – od przeglądarki, przez serwer aplikacji, aż po bazę danych i usługi zewnętrzne. Każda z tych warstw może zawierać potencjalną lukę, która – jeśli zostanie odpowiednio wykorzystana – pozwoli na przejęcie kontroli nad systemem lub ujawnienie danych.

Współczesne aplikacje nie są tworzone od podstaw – korzystają z gotowych frameworków, bibliotek, API oraz komponentów zewnętrznych. Z jednej strony przyspiesza to rozwój, z drugiej wprowadza dodatkowe ryzyka. Wystarczy jedna nieaktualna biblioteka, niedopatrzona zależność lub domyślna konfiguracja, by otworzyć podatny punkt dostępowy. Wraz z rozwojem architektur opartych na mikroserwisach i integracji z zewnętrznymi usługami, znacznie zwiększyła się tzw. powierzchnia ataku.

Nie mniej istotnym czynnikiem pozostaje czynnik ludzki. Wiele podatności nie wynika z problemów technicznych, lecz z błędów logicznych, pośpiechu, braku przeglądów kodu czy niedostatecznej świadomości zagrożeń. Nawet najbardziej zaawansowane narzędzia nie zastąpią wiedzy i dobrej praktyki programistycznej. Błędy często są powtarzalne i dotyczą podstawowych aspektów – takich jak nieprawidłowe uwierzytelnienie, brak walidacji danych, czy niewłaściwe zarządzanie sesją.

Bezpieczeństwo nie może być traktowane jako etap końcowy projektu – musi być obecne na każdym poziomie, od projektowania architektury systemu, przez implementację, aż po utrzymanie i monitoring działania aplikacji w środowisku produkcyjnym. Odpowiedzialność za nie ponoszą nie tylko specjaliści ds. bezpieczeństwa, ale cały zespół programistyczny, a także osoby odpowiedzialne za infrastrukturę i zarządzanie projektem.

Wzrost znaczenia aplikacji webowych niesie za sobą również zwiększone oczekiwania prawne i regulacyjne. Rozporządzenia takie jak RODO, standardy jak ISO 27001 czy wymagania branżowe w sektorze finansowym sprawiają, że naruszenia bezpieczeństwa mogą skutkować nie tylko stratami wizerunkowymi, ale też konkretnymi konsekwencjami prawnymi i finansowymi. Wymusza to nie tylko stosowanie odpowiednich narzędzi, ale także budowanie kultury bezpieczeństwa w organizacjach.

W tym kontekście coraz większego znaczenia nabierają inicjatywy, które pomagają porządkować wiedzę, promować dobre praktyki i standaryzować podejście do identyfikacji zagrożeń. Jedną z takich inicjatyw jest OWASP – międzynarodowa organizacja, która od lat tworzy ogólnodostępne zasoby edukacyjne i zestawienia najczęstszych błędów bezpieczeństwa w aplikacjach webowych. To właśnie jej działalności poświęcony jest kolejny podrozdział.

## OWASP – misja organizacji i wpływ na standardy bezpieczeństwa

W odpowiedzi na rosnącą potrzebę systematyzowania wiedzy z zakresu bezpieczeństwa aplikacji, w 2001 roku powstała organizacja OWASP (Open Worldwide Application Security Project). Jest to niezależna, międzynarodowa inicjatywa non-profit, której misją jest zwiększanie świadomości i promowanie dobrych praktyk w zakresie tworzenia bezpiecznego oprogramowania.

Jednym z głównych założeń OWASP jest pełna otwartość – wszystkie materiały, narzędzia oraz dokumenty publikowane przez organizację są udostępniane publicznie i bezpłatnie. Dzięki temu mogą z nich korzystać nie tylko eksperci ds. bezpieczeństwa, ale również programiści, testerzy, architekci systemów, a także studenci i osoby rozpoczynające swoją przygodę z tworzeniem aplikacji.

OWASP skupia się na dostarczaniu wiedzy w sposób zrozumiały i praktyczny. Oferuje poradniki, checklisty, zestawienia podatności, jak również narzędzia wspomagające testowanie bezpieczeństwa aplikacji. Projekty organizacji powstają w oparciu o współpracę społeczności – specjalistów z całego świata, dzielących się doświadczeniem oraz analizą rzeczywistych incydentów.

Wpływ OWASP na branżę IT jest znaczący. Tworzone przez nią dokumenty są szeroko stosowane w audytach, kursach, procesach rekrutacyjnych oraz jako odniesienie w politykach bezpieczeństwa organizacji. Co więcej, standardy OWASP są uwzględniane w wymaganiach regulatorów – m.in. w sektorze finansowym czy publicznym – co potwierdza ich istotność nie tylko techniczną, ale również formalno-prawną.

Najbardziej znanym projektem organizacji pozostaje OWASP Top 10 – lista najczęściej występujących i najistotniejszych podatności bezpieczeństwa w aplikacjach webowych. Nie jest to jednak jedyna inicjatywa OWASP. Wśród innych warto wymienić m.in. OWASP ASVS (Application Security Verification Standard), OWASP SAMM (Software Assurance Maturity Model) czy ZAP (Zed Attack Proxy) – popularne narzędzie do testów penetracyjnych.

OWASP nie definiuje konkretnych rozwiązań technologicznych ani nie narzuca implementacji. Zamiast tego promuje podejście oparte na świadomości ryzyk, rozpoznawaniu wzorców podatności i przeciwdziałaniu zagrożeniom poprzez dobre praktyki. To sprawia, że zasoby OWASP są uniwersalne i mogą być stosowane niezależnie od stosowanego języka programowania czy frameworka – również w środowisku Ruby on Rails, które jest przedmiotem dalszej części niniejszej pracy.

## Przegląd dokumentu OWASP Top 10 (2021)

OWASP Top 10 to flagowy projekt organizacji OWASP, który od lat stanowi jeden z najważniejszych standardów w zakresie bezpieczeństwa aplikacji webowych. Dokument ten przedstawia zestawienie najpowszechniejszych i najbardziej krytycznych podatności, które stanowią realne zagrożenie dla systemów informatycznych. Co istotne, lista ta nie powstaje w oparciu o teoretyczne rozważania, lecz bazuje na analizie rzeczywistych danych, pochodzących z tysięcy testów bezpieczeństwa przeprowadzanych przez organizacje na całym świecie.

Każda edycja OWASP Top 10 opiera się na trzech głównych źródłach: zgłoszeniach od społeczności, danych statystycznych oraz analizie trendów występujących w świecie rzeczywistym. Dzięki temu dokument jest nie tylko aktualny, ale również odzwierciedla ewoluujące zagrożenia oraz zmieniające się podejście do tworzenia oprogramowania.

Wersja opublikowana w 2021 roku przyniosła kilka istotnych zmian w stosunku do poprzednich edycji. Przede wszystkim rozszerzono zakres analizy – obok klasycznych błędów implementacyjnych pojawiły się także problemy o charakterze architektonicznym i projektowym, takie jak nieprzemyślana kontrola dostępu czy błędne założenia dotyczące integralności danych. W ten sposób OWASP podkreślił, że bezpieczeństwo nie zaczyna się w momencie pisania kodu, lecz znacznie wcześniej – już na etapie planowania aplikacji.

OWASP Top 10 pełni wiele ról jednocześnie. Z jednej strony stanowi narzędzie edukacyjne – pomaga nowym programistom zrozumieć, jakie błędy są najczęściej popełniane i jakie mają konsekwencje. Z drugiej – jest wykorzystywany jako podstawa do tworzenia polityk bezpieczeństwa, checklist w projektach informatycznych, a także jako punkt odniesienia przy audytach aplikacji. W wielu firmach i instytucjach finansowych dokument ten jest uznawany za standard minimalny, którego spełnienie jest wymagane przy wdrożeniach i testach.

Lista OWASP Top 10 nie jest zamkniętym katalogiem wszystkich możliwych zagrożeń. Jej celem jest skupienie uwagi na tych obszarach, które najczęściej prowadzą do realnych incydentów bezpieczeństwa. Dlatego też dokument ten nieustannie ewoluuje i stanowi punkt wyjścia, a nie kres, dla osób chcących rozwijać wiedzę w zakresie bezpiecznego programowania.

W dalszej części pracy omówione zostaną wybrane podatności z listy OWASP Top 10 (2021), wraz z przykładami ich występowania w środowisku Ruby on Rails oraz zaleceniami dotyczącymi ich unikania i eliminowania.

# Ruby on Rails

Ruby on Rails to webowy framework oparty na języku Ruby, zaprojektowany z myślą o szybkim i efektywnym tworzeniu nowoczesnych aplikacji internetowych. Powstał w 2004 roku i szybko zdobył popularność dzięki pragmatycznemu podejściu do projektowania oprogramowania oraz bogatemu zestawowi narzędzi dostępnych bezpośrednio po instalacji.

Framework upraszcza pracę programisty poprzez zestaw gotowych konwencji i domyślną strukturę projektu, co minimalizuje potrzebę ręcznej konfiguracji. Wbudowane mechanizmy takie jak routing, integracja z bazą danych czy generatory kodu znacząco przyspieszają proces tworzenia aplikacji. Dzięki temu Ruby on Rails jest często wybierany jako technologia startowa w projektach, w których liczy się czas dostarczenia pierwszej wersji produktu (**MVP**).

Na przestrzeni lat Rails ugruntował swoją pozycję szczególnie w środowisku startupowym oraz w firmach tworzących oprogramowanie typu SaaS. Mimo że jego popularność w Polsce jest stosunkowo niewielka, w krajach takich jak Stany Zjednoczone czy Wielka Brytania pozostaje cenionym wyborem wśród zespołów produktowych.

## Koncepcje architektoniczne i podejście do tworzenia aplikacji

Architektura Ruby on Rails opiera się na wzorcu **MVC** (Model–View–Controller), który dzieli aplikację na warstwy odpowiedzialne za dane, logikę biznesową oraz interfejs użytkownika. Takie podejście wspiera modularność, ułatwia testowanie oraz rozwój poszczególnych komponentów, co przekłada się na lepszą organizację i łatwiejsze utrzymanie kodu.

Rails promuje zasadę „**konwencji ponad konfigurację**”, dzięki czemu wiele decyzji projektowych zostało ustandaryzowanych. Programista korzysta z przewidywalnych nazw, struktury katalogów i domyślnych zachowań, co znacząco przyspiesza rozwój aplikacji. Uzupełnieniem tej filozofii jest zasada **DRY** (Don’t Repeat Yourself), która zachęca do reużywalności kodu poprzez mechanizmy takie jak helpery, partiale czy concerny, ograniczając duplikację i potencjalne błędy.

Ruby on Rails to framework oferujący kompletne środowisko do tworzenia aplikacji webowych już po instalacji, bez potrzeby dodatkowej konfiguracji. W standardzie zawiera:

* **Narzędzia programistyczne**: system routingu, generatory kodu (scaffolding), silnik widoków oraz wbudowany ORM (Active Record) do pracy z bazą danych.
* **Obsługę danych i bezpieczeństwa**: walidację danych, migracje schematów, mechanizmy sesji, ciasteczek i wbudowane zabezpieczenia.
* **Wsparcie dla różnych baz danych**: PostgreSQL, MySQL oraz SQLite — idealna do testów i prototypów.
* **Nowoczesne dodatki**: Hotwire (do tworzenia dynamicznych interfejsów bez JavaScript), Kamal (do wdrażania aplikacji), oraz komponenty Solid (kolejkowanie zadań, cache, WebSockety), które eliminują potrzebę korzystania z zewnętrznych usług, takich jak Redis.

Rails oferuje także gotowy plik **Dockerfile**, ułatwiający uruchamianie aplikacji w produkcyjnych środowiskach takich jak **Docker Compose** lub **Kubernetes**.

Taka kompletność pozwala budować rozbudowane aplikacje webowe bez konieczności integrowania wielu osobnych bibliotek czy narzędzi, co przekłada się na szybszy start projektu, większą spójność kodu i łatwiejsze utrzymanie całości.

## Ruby on Rails jako stabilne środowisko tworzenia i rozwijania aplikacji webowych

Ruby on Rails to dojrzały i sprawdzony framework do budowy aplikacji webowych, wykorzystywany przez światowe marki, takie jak **GitHub**, **Shopify**, **Airbnb** czy **GitLab**. Zaufanie tak różnorodnych firm potwierdza jego stabilność, niezawodność oraz odporność na dynamiczne zmiany w trendach technologicznych.

Z perspektywy biznesowej Rails oferuje **wyjątkowo szybki cykl rozwoju** – od pomysłu do działającego produktu. Framework umożliwia stworzenie pełnoprawnego MVP w bardzo krótkim czasie, co daje przewagę rynkową, pozwala testować rozwiązania, reagować na potrzeby użytkowników i elastycznie rozwijać produkt. Skrócony czas wdrożenia przekłada się bezpośrednio na niższe koszty i większą kontrolę nad projektem.

Jedną z kluczowych cech Ruby on Rails jest to, że dostarcza **kompletny zestaw narzędzi do tworzenia aplikacji** – od backendu, przez frontend, aż po zadania w tle i obsługę komunikacji w czasie rzeczywistym. Dzięki temu kompetentny programista może **samodzielnie** zbudować pełnoprawną aplikację, bez konieczności angażowania rozbudowanego zespołu. To rozwiązanie wyjątkowo korzystne dla startupów oraz firm działających z ograniczonym budżetem.

Framework wyróżnia się także **czytelnością kodu** – składnia języka Ruby przypomina język naturalny, co ułatwia komunikację pomiędzy programistami a osobami nietechnicznymi, takimi jak właściciele biznesów czy menedżerowie produktu. Kod jest intuicyjny, bardziej przewidywalny i dobrze oddaje intencje biznesowe, co przekłada się na łatwiejsze wdrażanie nowych członków zespołu i długoterminowe utrzymanie projektu.

Rails kładzie duży nacisk na **bezpieczeństwo** – wiele mechanizmów ochronnych jest aktywnych domyślnie. W kolejnych rozdziałach pracy omówione zostaną szczegółowo wbudowane zabezpieczenia chroniące przed najczęstszymi typami ataków.

**Silna i aktywna społeczność**, tysiące dostępnych bibliotek (tzw. gemów) oraz podejście oparte na konwencji sprawiają, że Rails eliminuje konieczność rozwiązywania typowych problemów technicznych. Programista może skupić się na rozwoju produktu, zamiast walczyć z frameworkiem.

Nauka Ruby on Rails to inwestycja, która zwraca się szybko. Wysoka produktywność, intuicyjność oraz bogate ekosystemy narzędzi sprawiają, że jest to technologia idealna dla firm stawiających na szybki rozwój, elastyczność i długoterminową stabilność.

# Struktura repozytorium z przykładami podatności

W ramach niniejszej pracy przygotowane zostało repozytorium zawierające zestaw przykładów podatności OWASP Top 10 (2021) w kontekście frameworka Ruby on Rails. Przykłady zostały podzielone według kategorii OWASP oraz konkretnych typów błędów, które zostały opisane i – tam, gdzie to możliwe – zaprezentowane w działającej aplikacji demonstracyjnej.

Repozytorium dostępne jest online pod adresem: <https://github.com/maciejb2k/ruby-on-rails-security>, oraz jako załącznik offline do pracy magisterskiej, w postaci skompresowanego folderu.

## Organizacja monorepozytorium z przykładami

Struktura repozytorium oparta została o monorepozytorium – każdy folder w katalogu głównym odpowiada jednej konkretnej podatności. W głównym katalogu repozytorium znajduje się plik README.md, pełniący rolę spisu treści i ułatwiający nawigację między przykładami.

Każdy podkatalog zawiera wszystkie niezbędne pliki do uruchomienia aplikacji: kod źródłowy, pliki konfiguracyjne oraz – w razie potrzeby – pliki pomocnicze do testowania.

Aplikacje uruchamiane są z wykorzystaniem Docker Compose. Katalogi projektów montowane są jako wolumeny do kontenerów Dockera, co umożliwia wygodne przeglądanie, modyfikowanie oraz testowanie kodu źródłowego. Przykładowe aplikacje zostały przygotowane w języku angielskim.

## Uruchamianie aplikacji testowych z wykorzystaniem Docker Compose

Aby uruchomić aplikację demonstracyjną, wymagane jest wcześniejsze zainstalowanie **Dockera**. Po pobraniu repozytorium należy przejść do katalogu odpowiadającego danej podatności, np.:

cd remote-code-execution

Następnie aplikacja może zostać uruchomiona poleceniem:

docker compose up -d

Zatrzymanie aplikacji odbywa się za pomocą:

docker compose down

Aby ponownie załadować dane testowe lub odświeżyć środowisko, można skorzystać z komendy:

docker compose restart

Aplikacje domyślnie działają na porcie 3000, dlatego przed uruchomieniem kolejnej instancji należy upewnić się, że poprzednia została wyłączona (np. poprzez docker compose down), aby uniknąć konfliktu portów. Po uruchomieniu, aplikacja będzie dostępna pod adresem:: <http://localhost:3000>

Wszystkie aplikacje zawierają predefiniowane dane testowe (seedy) zapisane w języku angielskim. Dane te są automatycznie odtwarzane po restarcie kontenerów, co umożliwia wielokrotne wykonywanie testów według zdefiniowanych scenariuszy.

W niektórych przypadkach dołączony został plik requests.http, zawierający gotowe zapytania HTTP służące do testowania podatności. Dla wygody zaleca się korzystanie z edytora Visual Studio Code z zainstalowanym rozszerzeniem REST Client, umożliwiającym wykonywanie żądań bezpośrednio z poziomu edytora.

# Analiza wybranych podatności z dokumentu OWASP Top 10 (2021) w kontekście Ruby on Rails

## A01: Broken Access Control

Kategoria A01 z listy OWASP Top 10 2021 dotyczy błędów kontroli dostępu, które umożliwiają nieuprawnionym użytkownikom wykonywanie działań poza ich przyznanymi uprawnieniami. W kontekście Ruby on Rails częstymi przyczynami tych podatności są nieprawidłowo skonfigurowane mechanizmy autoryzacji, brak weryfikacji własności obiektów lub nadmierne zaufanie do danych klienta. Skutki mogą obejmować nieautoryzowany dostęp do wrażliwych danych, modyfikację lub usunięcie zasobów, a nawet przejęcie kont administratora.

### Mass Assignment

#### Opis

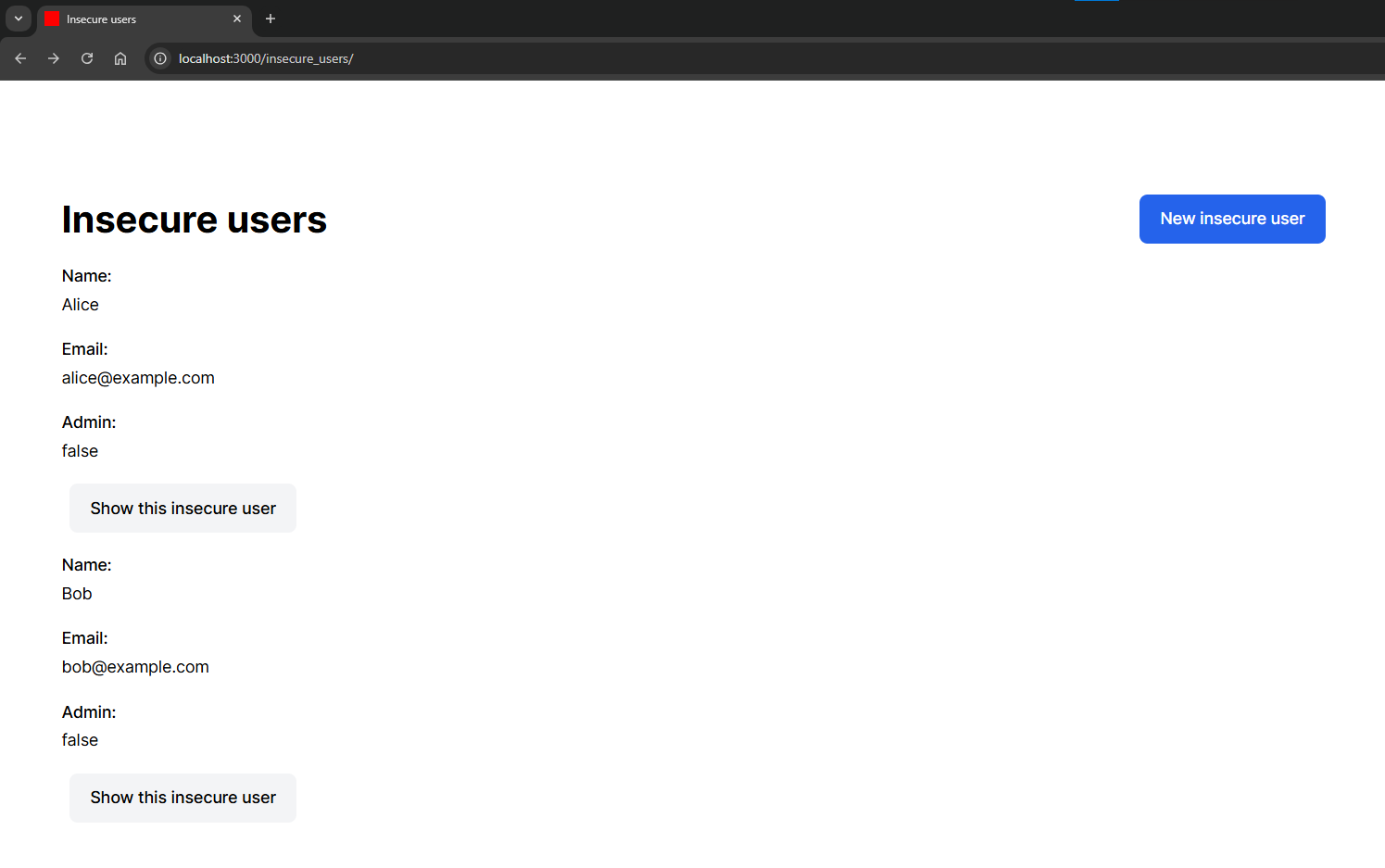
Mechanizm mass assignment w Ruby on Rails umożliwia zbiorcze przypisanie wartości atrybutów modelu na podstawie parametrów żądania HTTP, co stanowi wygodne rozwiązanie programistyczne podczas operacji tworzenia lub aktualizacji rekordów.

Podatność występuje, gdy aplikacja nie filtruje właściwie parametrów wejściowych, umożliwiając atakującemu modyfikację atrybutów, które powinny pozostać chronione (np. flagi administratora, uprawnienia dostępu czy wrażliwe dane użytkowników).

#### Przykład

Przykładowa aplikacja ilustrująca podatność typu mass assignment znajduje się w katalogu ruby-on-rails-security/mass-assignment w repozytorium projektowym dla pracy magisterskiej. Po uruchomieniu aplikacji należy przejść pod następujący adres w przeglądarce: <http://localhost:3000/insecure_users>.

Na poniższym rysunku przedstawiono listę użytkowników, których dane są obsługiwane przez model InsecureUser podatny na atak typu mass assignment.



Rysunek 1

Następnie wyświetlamy szczegóły pierwszego użytkownika z listy. Na poniższym rysunku możemy zaobserwować, że atrybut admin posiada wartość false, czyli w naszym przykładzie osoba **nie ma uprawnień administratora**.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 2

Na poniższym rysunku, w formularzu edycji danych tego użytkownika, **nie znajduje** się pole umożliwiające modyfikację atrybutu admin.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 3

Można jednak założyć, że potencjalny atakujący zna strukturę modelu i podejmie próbę bezpośredniej modyfikacji atrybutów tego użytkownika za pomocą specjalnie przygotowanego wcześniej żądania HTTP.

W pliku requests.http znajdującym się w katalogu projektu znajduje się przygotowane wcześniej do wysłania zapytanie http modyfikujące atrybuty tego użytkownika. Zapytanie HTTP można wykonać za pomocą dowolnego narzędzia np. REST Client w Visual Studio Code.

Na poniższym rysunku widzimy, że po wykonaniu zapytania aplikacja zwróciłą odpowiedź HTTP 200 OK, czyli rekord został zaktualizowany.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 4

Następnie należy ponownie otworzyć stronę użytkownika: Na poniższym rysunku widzimy, że zaktualizowana strona ze szczegółami użytkownika pokazuje, że wartość atrybutu admin została zmieniona na true, co potwierdza istnienie podatności związanej z masowym przypisaniem w modelu InsecureUser.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 5

#### Skutki

Podatność typu mass assignment może prowadzić do poważnych naruszeń bezpieczeństwa aplikacji, w tym między innymi:

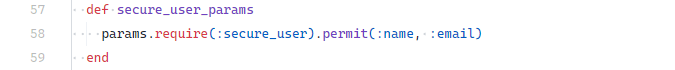
* Nieautoryzowanego nadania sobie podwyższonych uprawnień (np. rola administratora).
* Modyfikacji wrażliwych atrybutów, które nie powinny być edytowalne przez użytkownika.
* Przejęcia kontroli nad kontami lub zasobami innych użytkowników.

#### Zalecenia

##### Strong Parameters

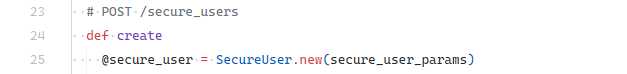
W celu zabezpieczenia aplikacji przed podatnością typu mass assignment należy stosować mechanizm Strong Parameters, który umożliwia jawne określenie dozwolonych parametrów w kontrolerze. Należy precyzyjnie definiować listę atrybutów możliwych do przypisania.

Przykład poprawnej implementacji, w której wykluczono atrybut admin, dzięki czemu nie może on zostać zmodyfikowany nawet w przypadku ręcznego wysłania żądania HTTP z próbą jego nadpisania.



Rysunek 6

Dzięki temu w metodach tworzących lub aktualizujących użytkowników, wywołujemy metodę z określonymi strong parameters i mamy pełną kontrolę nad atrybutami rekordu, tak jak to widać na poniższych fragmentach kodu z kontrolera:



Rysunek 7

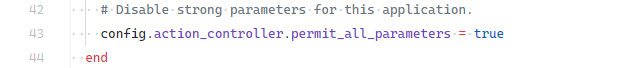
Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 8

##### Unikanie globalnego wyłączenia mechanizmu strong parameters

W przypadku próby przypisania atrybutów do modelu bez wykorzystania mechanizmu Strong Parameters, framework Ruby on Rails zgłosi wyjątek ActiveModel::ForbiddenAttributesError. Domyślnie ochrona ta jest aktywna. Na potrzeby aplikacji demonstracyjnej została ona celowo wyłączona w pliku config/application.rb, jednak w środowiskach produkcyjnych należy unikać takiego działania.



Rysunek 9

### Insecure Direct Object References (IDOR)

#### Opis

Podatność typu Insecure Direct Object Reference, polega na braku weryfikacji uprawnień do zasobów identyfikowanych za pomocą ID w adresie URL. Aplikacja pozwala użytkownikowi na dostęp do zasobów innych użytkowników poprzez ręczną modyfikację identyfikatora zadania, co jest wynikiem braku ograniczenia zapytań wyłącznie do danych należących do danego właściciela zasobu.

#### Przykład

Przykładowa aplikacja ilustrująca podatność typu Insecure Direct Object Reference znajduje się w katalogu ruby-on-rails-security/broken-access-control w repozytorium projektowym dla pracy magisterskiej. Po uruchomieniu aplikacji należy przejść pod następujący adres w przeglądarce <http://localhost:3000>.

Aplikacja umożliwia użytkownikom po wcześniejszym zalogowaniu się na zarządzanie swoimi zadaniami. W aplikacji występują dwaj użytkownicy, których dane do logowania są następujące:

* Użytkownik 1: maciek@example.com:password
* Użytkownik 2: tomek@example.com:password

Po zalogowaniu się jako użytkownik maciek@example.com, możemy zobaczyć listę jego wszystkich zadań, tak jak to widać na poniższym rysunku:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Strona internetowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 10

Gdy wejdziemy w szczegóły pierwszego zadania, widzimy, że należy ono na pewno do zalogowanego użytkownika i ma ono w bazie danych id=1, bo widać to na pasku URL na poniższym rysunku:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Strona internetowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 11

Co gdybyśmy spróbowali zmienić id zadania w adresie URL na 6? Modyfikując adres URL, jesteśmy w stanie zobaczyć zadanie należące do użytkownika [tomek@example.com](mailto:tomek@example.com), tak jak to widać na poniższym rysunku

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Strona internetowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 12

To oznacza, że aplikacja jest podatna na atak Insecure Direct Object Reference, ponieważ nie sprawdza, czy użytkownik ma dostęp do zasobu, do którego próbuje uzyskać dostęp.

Przyjrzyjmy się metodzie show, która odpowiada za wyświetlanie danego zadania użytkownikowi w kontrolerze app/controllers/tasks\_controller.rb.

Jak widać na poniższym rysunku, metoda ta nie sprawdza, czy zalogowany użytkownik ma dostęp do zasobu, do którego próbuje uzyskać dostęp, tylko przeszukuje wszystkie zadania należące do wszystkich użytkowników globalnie w całej tabeli.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 13

W celu zabezpieczenia aplikacji należy ograniczyć dostęp do zasobów wyłącznie dla ich właścicieli. Poprawnie zaimplementowana metoda show powinna wykorzystywać metodę current\_user, umożliwiającą pobranie jedynie zadań należących do aktualnie zalogowanego użytkownika.

Obraz zawierający tekst, zrzut ekranu, Czcionka, linia

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 14

Próba uzyskania dostępu do tego samego zadania innego użytkownika przy użyciu poprawnie zaimplementowanej metody secure\_show zakończy się niepowodzeniem – aplikacja zwróci błąd 404, tak jak na poniższym rysunku, ponieważ zadanie nie należy do aktualnie zalogowanego użytkownika.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 15

#### Skutki

Podatność typu Insecure Direct Object Reference może prowadzić do poważnych naruszeń bezpieczeństwa aplikacji, w tym między innymi:

• Nieautoryzowanego dostępu do danych innych użytkowników (np. ich zadań, profili lub dokumentów).

• Możliwości modyfikacji lub usunięcia cudzych zasobów bez odpowiednich uprawnień.

• Naruszenia poufności oraz integralności przechowywanych danych.

#### Zalecenia

W celu ochrony aplikacji przed tą podatnością należy zawsze ograniczać zapytania wyłącznie do **zasobów należących do właściwego użytkownika**. Można to osiągnąć poprzez wykorzystanie metody current\_user lub ręczne zawężenie zakresu operacji na danych, na przykład przez dodanie odpowiednich warunków w zapytaniu (np. where(user\_id: 1)).

Uzupełnieniem ochrony może być również ukrywanie wewnętrznych identyfikatorów zasobów poprzez zastosowanie identyfikatorów UUID zamiast prostych wartości liczbowych w adresach URL, co znacząco utrudnia ich odgadywanie przez osoby nieuprawnione.

### Object Level Authorization (OLA)

#### Opis

Podatność związana z brakiem kontroli dostępu na poziomie obiektów (Object Level Authorization) polega na nieweryfikowaniu, czy użytkownik posiada odpowiednie uprawnienia do wykonania operacji na konkretnym zasobie. Nawet jeśli użytkownik jest poprawnie uwierzytelniony, aplikacja może nie sprawdzać, czy dany zasób faktycznie należy do niego.

W przeciwieństwie do klasycznego przypadku Insecure Direct Object Reference (IDOR), ochrona przed tego typu podatnością polega nie tylko na filtrowaniu danych, ale również na stosowaniu dedykowanych mechanizmów autoryzacji na poziomie obiektów, takich jak **klasy polis**. W przypadku frameworka Ruby on Rails jednym z najpopularniejszych narzędzi wykorzystywanych do tego celu jest biblioteka **Pundit**.

#### Przykład

Przykładowa aplikacja ilustrująca podatność typu Object Level Authorization znajduje się w katalogu ruby-on-rails-security/broken-access-control w repozytorium projektowym dla pracy magisterskiej. Po uruchomieniu aplikacji należy przejść pod następujący adres w przeglądarce <http://localhost:3000>.

Po zalogowaniu się jako użytkownik maciek@example.com, wyświetlana jest lista przypisanych do niego zadań. Każde z nich posiada unikalny identyfikator w URL, co pozwala na jego ręczną modyfikację.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Strona internetowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 16

Podobnie jak w przypadku podatności IDOR, możliwe jest zmodyfikowanie adresu URL (np. zmiana /tasks/1 na /tasks/6) w celu uzyskania dostępu do zadania należącego do innego użytkownika. Jednakże w tym przypadku aplikacja zabezpieczona została mechanizmem autoryzacji na poziomie obiektów, wykorzystującym bibliotekę **Pundit**.

W pliku app/policies/task\_policy.rb zdefiniowano zasady autoryzacji dla poszczególnych akcji w kontrolerze (nazwy metod w klasie odpowiadają nazwom metod w kontrolerze):

Obraz zawierający tekst, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 17

W kontrolerze app/controllers/tasks\_controller.rb zastosowano wywołanie metody authorize, która uruchamia odpowiednią metodę polisy dla danego zasobu. W przypadku, gdy nazwa metody w klasie polisy odpowiada nazwie akcji w kontrolerze, możliwe jest pominięcie drugiego argumentu metody authorize, ponieważ domyślnie zostanie wywołana metoda odpowiadająca aktualnej akcji.

Na poniższym rysunku przykład wykorzystania dla edycji zadania użytkownika i bezpiecznego wyświetlania szczegółów zadania z wykorzystaniem polisy:

Obraz zawierający tekst, numer, oprogramowanie, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 18

W przypadku próby uzyskania dostępu do zasobu, który nie należy do zalogowanego użytkownika, aplikacja zwraca odpowiedź HTTP 403 Forbidden, tak jak na poniższym rysunku:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 19

#### Skutki

Brak kontroli dostępu na poziomie obiektów może prowadzić do poważnych naruszeń bezpieczeństwa aplikacji, w tym między innymi:

* Nieautoryzowanego podglądu, edycji lub usunięcia zasobów należących do innych użytkowników.
* Ominięcia logiki kontroli dostępu w aplikacjach złożonych z wielu warstw.
* Trwałego naruszenia integralności i poufności danych.

#### Zalecenia

W celu ochrony aplikacji przed tego typu podatnością należy stosować dedykowane mechanizmy autoryzacji, takie jak biblioteka Pundit, CanCanCan lub ActionPolicy, które umożliwiają definiowanie zasad dostępu w formie klas polisy.

Dobrą praktyką jest również stosowanie zasady "deny by default", polegającej na wymuszaniu jawnej autoryzacji każdej akcji kontrolera poprzez wywołania authorize oraz policy\_scope. Można to osiągnąć m.in. poprzez dodanie do bazowego kontrolera (np. ApplicationController) poniższych wywołań, widocznych na poniższym rysunku:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Stosowanie klas polisy nie tylko zwiększa bezpieczeństwo aplikacji, ale również poprawia czytelność i ułatwia jej dalsze utrzymanie. Należy unikać ręcznego sprawdzania uprawnień w kontrolerach na rzecz deklaratywnego podejścia. Dodatkowo zaleca się regularne testowanie scenariuszy nieautoryzowanego dostępu, w tym prób manipulacji parametrami w adresie URL.

### Local File Inclusion (LFI)

#### Opis

Podatność typu Local File Inclusion (LFI) polega na umożliwieniu atakującemu wczytania lokalnych plików znajdujących się na serwerze poprzez manipulację niezabezpieczonym wejściem użytkownika. W kontekście frameworka Ruby on Rails podatność ta może wystąpić w sytuacjach, gdy aplikacja dynamicznie ładuje pliki na podstawie wartości przekazywanych przez użytkownika, korzystając z metod takich jak render, send\_file czy File.read.

#### Przykład

Przykładowa aplikacja ilustrująca podatność typu Local File Inclusion znajduje się w katalogu ruby-on-rails-security/local-file-inclusion w repozytorium projektowym dla pracy magisterskiej. Po uruchomieniu aplikacji należy przejść pod następujący adres w przeglądarce <http://localhost:3000>.

W przygotowanej aplikacji demonstracyjnej, na stronie głównej dostępne są raporty finansowe zapisane lokalnie na serwerze, które użytkownik może pobrać. Podczas próby pobrania jednego z raportów aplikacja odczytuje plik z systemu plików serwera i udostępnia go użytkownikowi, co przedstawiono na poniższym rysunku.

Obraz zawierający oprogramowanie, Oprogramowanie multimedialne, tekst, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 20

Link do pobrania przykładowego sprawozdania finansowego wygląda następująco:

<http://localhost:3000/reports/unsafe_download?file=financial_report_q1.pdf>

W kontrolerze odpowiedzialnym za obsługę pobierania plików (reports\_controller.rb) znajduje się podatna logika odpowiadająca za realizację tej funkcjonalności.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 21

W powyższej implementacji parametr file przekazywany przez użytkownika nie jest w żaden sposób weryfikowany, co umożliwia pobranie dowolnego pliku z systemu plików serwera. Na przykład, modyfikując wartość parametru w adresie URL do ../../.env, możliwe jest uzyskanie dostępu do pliku .env zawierającego wrażliwe dane konfiguracyjne:

<http://localhost:3000/reports/unsafe_download?file=../../.env>

Analogicznie, możliwe jest odczytanie innych plików systemowych, takich jak /etc/passwd czy /etc/hosts.

Rozwiązaniem problemu jest weryfikacja, czy wskazany przez użytkownika plik znajduje się w dozwolonym katalogu, np. reports. W praktyce oznacza to konieczność sprawdzenia, czy ścieżka do pliku nie wykracza poza określony katalog roboczy. Kluczowym elementem zabezpieczenia jest filtrowanie oraz walidacja parametrów wejściowych. Przykład bezpiecznej implementacji pobierania plików przedstawiono poniżej:

Obraz zawierający tekst, Czcionka, numer, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

W przypadku próby pobrania pliku spoza dozwolonego katalogu korzystając z bezpiecznej metody, aplikacja zwraca odpowiedź 403 Forbidden, na przykład:

<http://localhost:3000/reports/safe_download?file=../../.env>

Obraz zawierający zrzut ekranu, tekst

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 22

#### Skutki

Podatność typu Local File Inclusion może prowadzić do poważnych naruszeń bezpieczeństwa aplikacji, w tym między innymi:

* Odczytu wrażliwych plików systemowych lub konfiguracyjnych (np. /etc/passwd, plików z danymi uwierzytelniającymi).
* Ujawnienia poufnych informacji, takich jak klucze API, dane użytkowników czy hasła.
* Eskalacji ataku do zdalnego wykonania kodu (Remote Code Execution) w przypadku wczytywania plików zawierających kod programistyczny.

#### Zalecenia

W celu zabezpieczenia aplikacji przed tą podatnością, należy stosować odpowiednią walidację i sanityzację parametrów wejściowych, w szczególności ograniczając możliwość odwoływania się do plików spoza dozwolonych katalogów.

Zaleca się unikanie dynamicznego budowania ścieżek na podstawie danych dostarczanych przez użytkownika bez uprzedniej walidacji, a zamiast tego wykorzystywanie pełnych, bezwzględnych ścieżek do konkretnych zasobów.

W przypadku korzystania z metod takich jak send\_file, należy bezwzględnie unikać bezpośredniego przekazywania do nich parametrów pochodzących od użytkownika. Dobrym rozwiązaniem jest zastosowanie tzw. białych list (ang. whitelisting), definiujących dozwolone pliki lub katalogi, do których może mieć dostęp użytkownik.

## A02:2021 - Cryptographic Failures

Kategoria A02:2021 Cryptographic Failures odnosi się do podatności wynikających z niewłaściwego stosowania mechanizmów kryptograficznych, które powinny chronić poufność, integralność i autentyczność danych. Błędy tego typu obejmują m.in. brak szyfrowania danych wrażliwych, wykorzystywanie przestarzałych lub słabych algorytmów szyfrujących, przechowywanie haseł bez odpowiedniego haszowania, stosowanie twardo zakodowanych kluczy kryptograficznych (hard-coded keys) czy przesyłanie danych w postaci jawnej przez niezabezpieczone połączenia. W kontekście Ruby on Rails błędy kryptograficzne mogą pojawić się na przykład wtedy, gdy aplikacja przechowuje hasła bez użycia bezpiecznego algorytmu haszującego (np. bcrypt), korzysta z niezalecanych funkcji takich jak MD5, SHA1, lub przekazuje dane logowania przez niezabezpieczone połączenie HTTP.

### Weak Encoding for Password

#### Opis

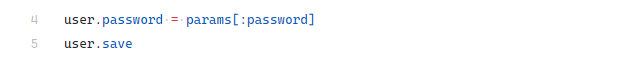
Podatność związana ze słabym kodowaniem haseł polega na nieprawidłowym przechowywaniu danych uwierzytelniających użytkowników, co znacząco obniża poziom bezpieczeństwa aplikacji. Nieodpowiednie techniki haszowania lub ich całkowity brak mogą prowadzić do przejęcia kont przez osoby nieuprawnione w wyniku ataków słownikowych, brute-force lub przy użyciu tęczowych tablic (rainbow tables).

#### Przykłady

##### Przechowywanie haseł w postaci jawnej

Jednym z najpopularniejszych błędów związanych z tą podatnością jest bezpośrednie zapisywanie hasła w bazie danych bez uprzedniego haszowania.

W poniższym przykładzie hasło użytkownika jest przypisywane wprost na podstawie danych z formularza, np. w kontrolerze, tak jak to widać na poniższym rysunku:



Rysunek 23

W takim przypadku, w razie wycieku bazy danych, atakujący uzyskuje dostęp do pełnej listy haseł w postaci jawnej, co może prowadzić do masowego przejęcia kont.

##### Użycie przestarzałych algorytmów hashujących

Innym błędem jest stosowanie niezalecanych obecnie i niebezpiecznych algorytmów haszujących, takich jak np. MD5, które nie zapewniają odpowiedniego poziomu bezpieczeństwa. Przykład błędnej implementacji może wyglądać następująco:

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

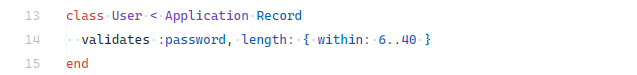
Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 24

MD5 jest w obecnych czasach wyjątkowo podatny na ataki tęczowych tablic i brute-force, co sprawia, że hasła mogą być łatwo odczytane.

##### Brak wymuszenia silnych haseł

Nawet przy poprawnym hashowaniu, stosowanie zbyt łagodnej polityki haseł znacząco obniża poziom zabezpieczeń. Poniższy przykład walidacji dopuszcza bardzo słabe hasła:



Minimalna zalecana długość hasła powinna wynosić **co najmniej 12 znaków**, przy czym dłuższe i zróżnicowane hasła zapewniają wyższy poziom bezpieczeństwa.

#### Skutki

Nieprawidłowe haszowanie lub przechowywanie haseł może prowadzić do poważnych konsekwencji bezpieczeństwa, w tym między innymi:

* Naruszenia prywatności użytkowników – w przypadku wycieku bazy danych, hasła przechowywane w postaci jawnej mogą zostać łatwo odczytane przez osoby nieuprawnione.
* Umożliwienia ataków typu brute-force i słownikowych – zastosowanie słabych lub przestarzałych algorytmów haszujących znacząco ułatwia łamanie haseł.
* Utratę reputacji i konsekwencje prawne – incydenty związane z wyciekiem danych mogą skutkować sankcjami prawnymi oraz utratą zaufania ze strony klientów.

#### Zalecenia

##### Używanie sprawdzonych rozwiązań do uwierzytelniania

Devise to jedna z najpopularniejszych bibliotek do autoryzacji w Ruby on Rails. Oferuje gotowe mechanizmy uwierzytelniania, w tym obsługę sesji, resetowania haseł, a także wsparcie dla dwuetapowej weryfikacji (2FA). Domyślnie wykorzystuje algorytm bcrypt do bezpiecznego przechowywania haseł.

Korzystanie z wyspecjalizowanych bibliotek, takich jak Devise, jest kluczowe z punktu widzenia bezpieczeństwa – samodzielne implementowanie mechanizmów uwierzytelniania wiąże się z dużym ryzykiem błędów, które mogą prowadzić do poważnych luk i naruszeń bezpieczeństwa.

##### Stosowanie has\_secure\_password

W przypadku prostszych aplikacji, które nie wymagają rozbudowanej funkcjonalności, warto skorzystać z natywnego rozwiązania dostępnego w Ruby on Rails – modułu has\_secure\_password. Jest to wbudowany mechanizm, który można łatwo zastosować w modelu użytkownika, tak jak to widać na poniższym rysunku:



Rysunek 25

Moduł ten zapewnia bezpieczne haszowanie haseł przy użyciu algorytmu bcrypt oraz automatyczną walidację ich obecności. Do jego działania wymagane jest jedynie dodanie kolumny password\_digest w tabeli użytkowników. Po aktywacji Rails samodzielnie obsługuje proces uwierzytelniania, co pozwala uniknąć ręcznej implementacji wrażliwych operacji.

Dodatkowo moduł automatycznie dodaje do modelu metodę authenticate, która służy do weryfikacji poprawności podanego hasła. Przykład logowania w kontrolerze:Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 26

### Use of Hard-Coded Cryptographic Key

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### Cleartext Transmission of Sensitive Information

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

## A03:2021 – Injection

### Remote Code Execution (RCE)

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### Cross-Site Scripting (XSS)

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### Cross-Site Request Forgery (CSRF)

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### SQL Injection (SQLi)

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

## A04:2021 - Insecure Design

### Open Redirect

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### Regex Denial of Service (ReDoS)

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### Login Rate Limiting

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

## A05:2021 - Security Misconfiguration

### Token / Cookie Misconfiguration

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### XML XXE

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### TLS Force / HSTS

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### Debug Mode

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

## A09:2021 - Security Logging and Monitoring Failures

### Enable Logging

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### Logging Sensitive Information

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

### No Log to User

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

## A10:2021 - Server-Side Request Forgery

### Server-Side Request Forgery

#### Opis

#### Przykład

#### Skutki

#### Zalecenia

# Zakończenie

# Bibliografia

# Wykaz rysunków

# Wykaz zdjęć i grafik z Internetu użytych w pracy

**Temat pracy w języku polskim**

Bezpieczeństwo aplikacji webowych w Ruby on Rails

**Temat pracy w języku angielskim**

Ruby on Rails security for web apps

**Streszczenie**

Lorem Ipsum dolor sit amet.

Obraz zawierający tekst, zrzut ekranu, Czcionka, list

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.