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](#_Toc195952264)

[1.1 Problematyka pracy 4](#_Toc195952265)

[1.2 Cel i zakres pracy 5](#_Toc195952266)

[1.3 Uzasadnienie innowacyjności tematyki 6](#_Toc195952267)

[2 Bezpieczeństwo aplikacji webowych 7](#_Toc195952268)

[2.1 OWASP – misja organizacji i wpływ na standardy bezpieczeństwa 7](#_Toc195952269)

[2.2 Przegląd dokumentu OWASP Top 10 (2021) 8](#_Toc195952270)

[3 Ruby on Rails 10](#_Toc195952271)

[3.1 Koncepcje architektoniczne i podejście do tworzenia aplikacji 10](#_Toc195952272)

[3.2 Ruby on Rails jako stabilne środowisko tworzenia i rozwijania aplikacji webowych 11](#_Toc195952273)

[4 Struktura repozytorium z przykładami podatności 13](#_Toc195952274)

[4.1 Organizacja monorepozytorium z przykładami 13](#_Toc195952275)

[4.2 Uruchamianie aplikacji testowych z wykorzystaniem Docker Compose 13](#_Toc195952276)

[5 Analiza wybranych podatności z dokumentu OWASP Top 10 (2021) w kontekście Ruby on Rails 15](#_Toc195952277)

[5.1 A01: Broken Access Control 15](#_Toc195952278)

[5.1.1 Mass Assignment 15](#_Toc195952279)

[5.1.2 Insecure Direct Object References (IDOR) 20](#_Toc195952280)

[5.1.3 Object Level Authorization (OLA) 23](#_Toc195952281)

[5.1.4 Local File Inclusion (LFI) 28](#_Toc195952282)

[5.2 A02:2021 - Cryptographic Failures 31](#_Toc195952283)

[5.2.1 Weak Encoding for Password 31](#_Toc195952284)

[5.2.2 Use of Hard-Coded Cryptographic Key 34](#_Toc195952285)

[5.2.3 Cleartext Transmission of Sensitive Information 37](#_Toc195952286)

[5.3 A03:2021 – Injection 43](#_Toc195952287)

[5.3.1 Remote Code Execution (RCE) 43](#_Toc195952288)

[5.3.2 Cross-Site Scripting (XSS) 48](#_Toc195952289)

[5.3.3 Cross-Site Request Forgery (CSRF) 55](#_Toc195952290)

[5.3.4 SQL Injection (SQLi) 61](#_Toc195952291)

[5.4 A04:2021 - Insecure Design 67](#_Toc195952292)

[5.4.1 Open Redirect 67](#_Toc195952293)

[5.4.2 Regex Denial of Service (ReDoS) 71](#_Toc195952294)

[5.4.3 Login Rate Limiting 74](#_Toc195952295)

[5.5 A05:2021 - Security Misconfiguration 78](#_Toc195952296)

[5.5.1 Token / Cookie Misconfiguration 78](#_Toc195952297)

[5.5.2 XML XXE 82](#_Toc195952298)

[5.5.3 TLS Force / HSTS 85](#_Toc195952299)

[5.5.4 Debug Mode 88](#_Toc195952300)

[5.6 A09:2021 - Security Logging and Monitoring Failures 89](#_Toc195952301)

[5.6.1 Enable Logging 89](#_Toc195952302)

[5.6.2 Logging Sensitive Information 93](#_Toc195952303)

[5.6.3 No Log to User 96](#_Toc195952304)

[5.7 A10:2021 - Server-Side Request Forgery 98](#_Toc195952305)

[6 Zakończenie 103](#_Toc195952306)

[Bibliografia 104](#_Toc195952307)

[Wykaz rysunków 104](#_Toc195952308)

[Wykaz zdjęć i grafik z Internetu użytych w pracy 104](#_Toc195952309)

# 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 fundament nowoczesnych systemów informatycznych, obsługując kluczowe obszary, od finansów po edukację i administrację. Ich globalna dostępność i skalowalność, choć przynoszą wiele korzyści, jednocześnie czynią je atrakcyjnym celem ataków. W przeciwieństwie do aplikacji lokalnych, systemy webowe są stale narażone na interakcje z zewnętrznym środowiskiem, a dane przepływają przez złożoną strukturę warstw, z których każda może zawierać potencjalne słabości.

Współczesny proces tworzenia aplikacji opiera się na wykorzystaniu gotowych komponentów, co z jednej strony przyspiesza rozwój, ale z drugiej wprowadza dodatkowe ryzyka związane z nieaktualnymi bibliotekami czy błędami konfiguracji. Nie bez znaczenia pozostaje także czynnik ludzki – wiele podatności wynika z błędów logicznych i niedopatrzeń programistycznych. Bezpieczeństwo nie może być traktowane jako dodatek, lecz musi być integralną częścią każdego etapu projektu.

W kontekście rosnących wymagań prawnych i regulacyjnych, dbałość o bezpieczeństwo aplikacji webowych staje się nie tylko kwestią techniczną, ale również prawną i biznesową. Inicjatywy takie jak OWASP odgrywają kluczową rolę w standaryzacji wiedzy i promowaniu najlepszych praktyk w zakresie identyfikacji i przeciwdziałania zagrożeniom, co jest niezwykle istotne w dynamicznie zmieniającym się krajobrazie cyberbezpieczeństwa.

## 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 **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 i wspomnianych już wcześniej 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

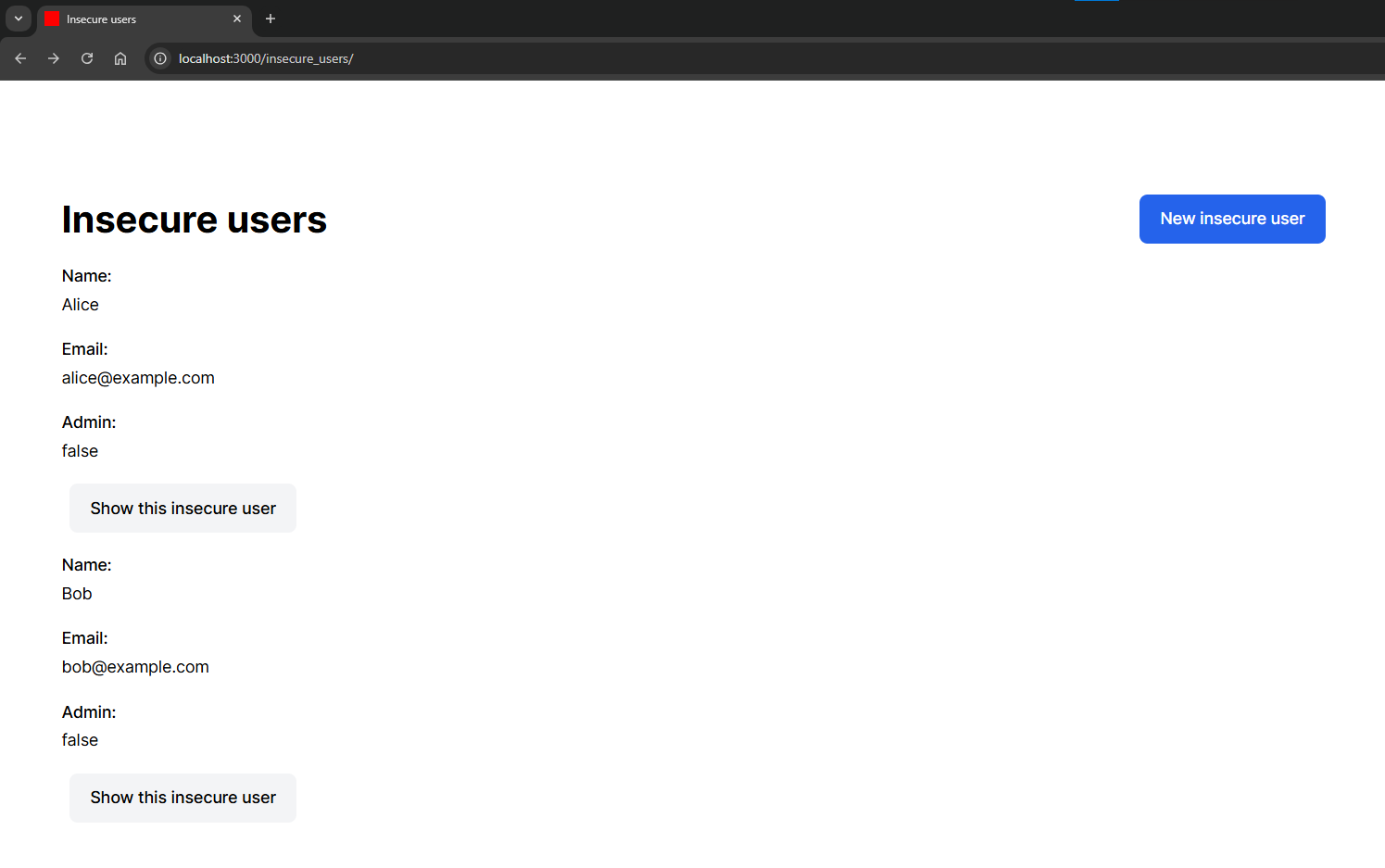
Mechanizm mass assignment w Ruby on Rails umożliwia zbiorcze przypisanie wartości atrybutów modelu na podstawie parametrów żądania HTTP, co stanowi wygodne rozwiązanie programistyczne podczas operacji tworzenia lub aktualizacji rekordów.

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:

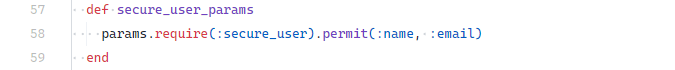
* Nieautoryzowanego nadania sobie podwyższonych uprawnień (np. rola administratora).
* Modyfikacji wrażliwych atrybutów, które nie powinny być edytowalne przez użytkownika.
* Przejęcia kontroli nad kontami lub zasobami innych użytkowników.

#### Zalecenia

##### Strong Parameters

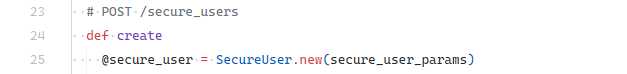
W celu zabezpieczenia aplikacji przed podatnością typu mass assignment należy stosować mechanizm Strong Parameters, który umożliwia jawne określenie dozwolonych parametrów w kontrolerze. Należy precyzyjnie definiować listę atrybutów możliwych do przypisania.

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

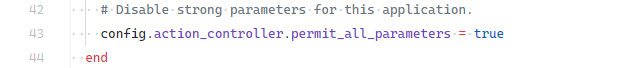
Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

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

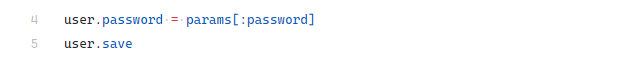
Podatność związana ze słabym kodowaniem haseł polega na nieprawidłowym przechowywaniu danych uwierzytelniających użytkowników, co znacząco obniża poziom bezpieczeństwa aplikacji. Nieodpowiednie techniki haszowania lub ich całkowity brak mogą prowadzić do przejęcia kont przez osoby nieuprawnione w wyniku 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

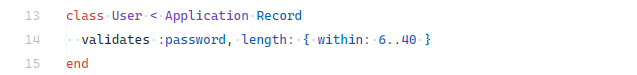
Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

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

Podatność typu Use of Hard-Coded Cryptographic Key polega na umieszczaniu kluczy kryptograficznych bezpośrednio w kodzie źródłowym aplikacji. Klucze te są często wykorzystywane do szyfrowania danych, podpisywania ciasteczek sesyjnych lub generowania tokenów uwierzytelniających. W przypadku ich ujawnienia – na przykład poprzez publiczne repozytorium – możliwe jest przejęcie sesji, fałszowanie żądań lub eskalacja uprawnień przez osoby nieuprawnione.

#### Przykład

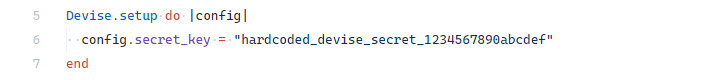
Jednym z typowych przypadków występowania tej podatności jest niewłaściwe zarządzanie tajnym kluczem wymaganym w bibliotece **Devise**, wykorzystywanej do uwierzytelniania użytkowników w aplikacjach Ruby on Rails.

**Devise** zapewnia gotowe mechanizmy obsługi sesji, logowania, rejestracji oraz resetowania haseł. W celu prawidłowego działania biblioteka wymaga zdefiniowania wartości Devise.secret\_key w pliku konfiguracyjnym config/initializers/devise.rb.

Klucz ten wykorzystywany jest m.in. do:

* podpisywania ciasteczek sesyjnych,
* generowania tokenów autoryzacyjnych (np. do resetowania hasła),
* zapewnienia integralności mechanizmów logowania i opcji „remember me”.

W przypadku podatnej aplikacji klucz ten może zostać zapisany bezpośrednio w pliku konfiguracyjnym, co przedstawia poniższy fragment kodu:



Rysunek 27

Umieszczenie klucza na stałe w kodzie źródłowym – szczególnie w repozytorium publicznym – stanowi poważne zagrożenie. W przypadku jego wycieku atakujący może wygenerować własne, poprawnie podpisane ciasteczka sesyjne, co w efekcie pozwala na przejęcie kont użytkowników bez znajomości ich danych uwierzytelniających.

#### Skutki

Wyciek twardo zakodowanych kluczy kryptograficznych w aplikacji Ruby on Rails może prowadzić do poważnych naruszeń bezpieczeństwa, obejmujących utratę poufności, integralności danych oraz kontroli nad mechanizmami autoryzacyjnymi. Atakujący, dysponując ujawniony kluczem, mogą:

* generować fałszywe tokeny sesyjne i uwierzytelniające,
* podpisywać własne żądania w sposób zgodny z oczekiwaniami aplikacji,
* odszyfrowywać poufne dane przechowywane w bazie,
* uzyskać nieautoryzowany dostęp do kont użytkowników lub zewnętrznych systemów.

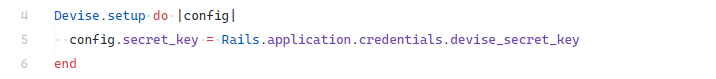
W przypadku, gdy klucze te dotyczą danych wrażliwych (np. haseł lub informacji finansowych), możliwe jest ich odszyfrowanie i dalsze wykorzystanie w celach nieuprawnionych. Upublicznienie takich sekretów – np. poprzez repozytorium kodu – naraża aplikację na natychmiastowe ataki, a w skrajnych przypadkach może doprowadzić do całkowitego przejęcia kontroli nad systemem.

#### Zalecenia

##### Unikanie przechowywania kluczy w kodzie źródłowym

Sekretne klucze nie powinny być zapisywane bezpośrednio w plikach źródłowych ani konfiguracyjnych wersjonowanych w repozytorium. Zaleca się korzystanie z mechanizmów takich jak **Rails Credentials** (config/credentials.yml.enc) lub **zmienne środowiskowe**, które umożliwiają dynamiczne pobieranie wartości w trakcie działania aplikacji.

Przykład poprawnego użycia Rails Credentials:



Rysunek 28

Przykład użycia zmiennej środowiskowej:

Obraz zawierający tekst, zrzut ekranu, Czcionka, biały

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 29

##### Regularna rotacja kluczy i unieważnianie sesji

Wszystkie klucze wykorzystywane do podpisywania sesji lub tokenów uwierzytelniających powinny być okresowo rotowane. W przypadku podejrzenia ich wycieku należy natychmiast unieważnić wszystkie aktywne sesje użytkowników i wygenerować nowe wartości kluczy.

##### Poprawne zarządzanie zmiennymi środowiskowymi

Podczas korzystania ze zmiennych środowiskowych należy zadbać o ich przechowywanie poza kodem aplikacji, np. w pliku .env (lokalnie) lub bezpośrednio w konfiguracji serwera (np. w systemie CI/CD lub usługach chmurowych). Należy upewnić się, że pliki z takimi danymi nie trafiają do repozytorium.

##### Monitorowanie i wykrywanie wycieków

Aby zapobiec przypadkowemu ujawnieniu sekretów, warto stosować automatyczne narzędzia do skanowania repozytoriów pod kątem poufnych danych, takie jak **GitLeaks** lub **TruffleHog**. Regularne skanowanie kodu pozwala szybko wykryć potencjalne naruszenia i ograniczyć ich skutki.

### Cleartext Transmission of Sensitive Information

#### Opis

Podatność typu *Cleartext Transmission of Sensitive Information* odnosi się do przesyłania wrażliwych danych w formie niezaszyfrowanej pomiędzy aplikacją Ruby on Rails a innymi komponentami systemu. Problem ten nie dotyczy wyłącznie komunikacji z przeglądarką użytkownika (gdzie standardowo stosuje się HTTPS), ale również połączeń wewnętrznych, takich jak dostęp do baz danych, systemów kolejkowych, usług SMTP czy zewnętrznych interfejsów API.

Skala ryzyka związanego z przesyłaniem danych w postaci otwartego tekstu jest ściśle zależna od środowiska, w którym działa aplikacja. W prostych wdrożeniach na serwerze VPS, gdzie wszystkie komponenty znajdują się na jednym hoście, ryzyko może być relatywnie niskie. W środowiskach bardziej złożonych – takich jak klastry Kubernetes, infrastruktura chmurowa lub rozwiązania hybrydowe – komunikacja często odbywa się pomiędzy fizycznie oddzielonymi serwerami i sieciami, co zwiększa ryzyko przechwycenia danych przez osoby nieuprawnione. Dodatkowo, w przypadku korzystania z usług zarządzanych przez różnych dostawców (np. aplikacja działa w chmurze AWS, ale baza danych znajduje się w zewnętrznej usłudze typu DBaaS), dane mogą być przesyłane przez sieć publiczną, jeśli nie zostanie zapewnione odpowiednie szyfrowanie połączeń.

Do najczęstszych przykładów nieprawidłowej konfiguracji należą brak wymuszenia szyfrowania w połączeniach z bazami danych (np. PostgreSQL), usługami typu Redis, brokerami wiadomości (np. RabbitMQ) czy integracjami z zewnętrznymi usługami poprzez klienty HTTP.

#### Przykłady

##### Połączenie z bazą danych PostgreSQL

W aplikacjach Ruby on Rails połączenie z bazą danych PostgreSQL można skonfigurować w pliku config/database.yml lub za pomocą zmiennej środowiskowej DATABASE\_URL. W przypadku braku odpowiednich parametrów szyfrowania dane przesyłane pomiędzy aplikacją a serwerem bazy mogą być przekazywane w postaci niezaszyfrowanej.

Przykład połączenia niewykorzystującego szyfrowania:



Rysunek 30

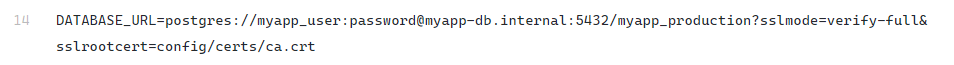
Aby wymusić szyfrowanie, należy użyć parametru sslmode.

sslmode=require – szyfruje transmisję, lecz nie weryfikuje certyfikatu serwera, przez co jest podatne na ataki typu *Man-in-the-Middle*.



Rysunek 31

sslmode=verify-full – zapewnia pełne szyfrowanie oraz weryfikację tożsamości serwera na podstawie certyfikatu zaufanego urzędu (CA). Wymaga dostarczenia pliku z certyfikatem:



Rysunek 32

W środowiskach lokalnych lub prostych wdrożeniach (np. VPS) ryzyko może być ograniczone, jeśli komponenty działają na tym samym hoście (localhost). Natomiast w architekturach rozproszonych – takich jak Kubernetes lub rozwiązania chmurowe – brak szyfrowania w komunikacji z bazą danych stanowi poważne zagrożenie.

Warto podkreślić, że większość zarządzanych usług baz danych (np. Amazon RDS, Heroku, DigitalOcean Managed Databases) domyślnie wymusza szyfrowanie i udostępnia odpowiednie certyfikaty CA do weryfikacji po stronie klienta.

##### Redis

Redis to popularna baza typu key-value, wykorzystywana w aplikacjach Rails m.in. jako magazyn sesji, kolejka zadań (np. Sidekiq) lub cache. Domyślnie Redis nie stosuje szyfrowania – połączenia redis:// przesyłają dane (w tym poświadczenia) w otwartym tekście.

Przykład połączenia nieszyfrowanego:



Rysunek 33

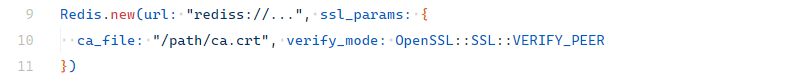
Brak szyfrowania jest akceptowalny w środowiskach lokalnych lub zamkniętych sieciach, szczególnie gdy Redis służy wyłącznie jako cache niewrażliwych danych. W przypadku przechowywania sesji użytkowników lub innych danych poufnych zaleca się stosowanie połączenia TLS (rediss://).

Połączenie szyfrowane bez weryfikacji certyfikatu (np. w usługach chmurowych):



Rysunek 34

Najbezpieczniejszą opcją jest połączenie z weryfikacją certyfikatu serwera:



Rysunek 35

Warto jednak rozważyć kompromis pomiędzy bezpieczeństwem a wydajnością – w niektórych przypadkach (np. Redis uruchomiony na tym samym hoście lub w tej samej podsieci co aplikacja) szyfrowanie może być zbędne i prowadzić do niepotrzebnych opóźnień.

##### SMTP

W aplikacjach Ruby on Rails brak wymuszenia szyfrowania połączenia z serwerem SMTP może prowadzić do przesyłania danych uwierzytelniających oraz treści wiadomości w postaci niezaszyfrowanej. Problem ten dotyczy zwłaszcza środowisk lokalnych, systemów legacy lub własnych serwerów pocztowych w sieciach wewnętrznych, gdzie nie skonfigurowano poprawnie TLS/STARTTLS.

Przykład niezaszyfrowanego połączenia:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 36

W celu zapewnienia bezpieczeństwa transmisji należy wymusić nawiązywanie szyfrowanego połączenia i włączyć opcję STARTTLS:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 37

W przypadku korzystania z zewnętrznych usług pocztowych (np. SendGrid, Mailgun), szyfrowanie jest zazwyczaj wymagane domyślnie. W środowiskach zamkniętych należy każdorazowo ocenić poziom zaufania do infrastruktury i odpowiednio dostosować konfigurację.

##### Integracje z API

W aplikacjach Ruby on Rails do komunikacji HTTP z innymi usługami (wewnętrznymi i zewnętrznymi) często wykorzystywana jest biblioteka **Faraday**.

Dla zewnętrznych API wymagane jest zawsze szyfrowanie z weryfikacją certyfikatu; w przypadku połączeń wewnętrznych zasadność użycia TLS zależy od poziomu izolacji środowiska i charakteru przesyłanych danych.

W przypadku **integracji zewnętrznych** (np. API partnerów, bramki płatności) połączenia muszą być realizowane przez HTTPS z aktywną weryfikacją certyfikatu:



Rysunek 38

Dla **połączeń wewnętrznych**, np. pomiędzy usługami w tym samym klastrze Kubernetes, brak TLS może być akceptowalny, jeśli środowisko jest w pełni zaufane:



Rysunek 39

Jeśli jednak ruch sieciowy przechodzi pomiędzy klastrami lub pomiędzy usługami w obrębie chmury publicznej, warto rozważyć HTTPS z własnym CA:



Rysunek 40

#### Skutki

Brak odpowiedniego szyfrowania transmisji danych w aplikacjach Ruby on Rails może prowadzić do szeregu zagrożeń, w tym:

* Przejęcia poświadczeń lub wrażliwych danych użytkowników w wyniku przesyłania ich w niezaszyfrowanej formie.
* Podatności na ataki typu Man-in-the-Middle, umożliwiające przechwycenie lub modyfikację komunikacji między usługami.
* Naruszenia obowiązujących regulacji prawnych (np. RODO, PCI-DSS), skutkujących odpowiedzialnością administracyjną lub finansową.

#### Zalecenia

##### Stosuj szyfrowanie TLS tam, gdzie to uzasadnione

We wszystkich połączeniach między aplikacją Rails a usługami (baza danych, Redis, SMTP, API) zaleca się stosowanie TLS, szczególnie w środowiskach produkcyjnych lub publicznych. W zamkniętych środowiskach (np. lokalne klastry) szyfrowanie może być opcjonalne, jeśli ryzyko przechwycenia ruchu jest znikome.

##### Weryfikuj certyfikaty serwerów

Konfiguracja klientów powinna zawsze uwzględniać weryfikację tożsamości serwera, np. sslmode=verify-full w PostgreSQL, VERIFY\_PEER w Redisie, enable\_starttls\_auto: true w SMTP. Brak tej weryfikacji otwiera drogę do ataków typu Man-in-the-Middle.

##### Korzystaj z usług zarządzanych, jeśli to możliwe

Usługi chmurowe (np. Amazon RDS, Heroku Redis, Google Cloud SQL) zwykle wymuszają transmisję zaszyfrowaną i dostarczają odpowiednie certyfikaty CA. Wybór takich rozwiązań zmniejsza ryzyko błędów konfiguracyjnych i ułatwia utrzymanie bezpieczeństwa.

##### Audytuj zmienne środowiskowe

Warto regularnie przeglądać zmienne takie jak DATABASE\_URL, REDIS\_URL czy SMTP\_URL, by upewnić się, że konfiguracja wymusza użycie szyfrowania (rediss://, sslmode=verify-full itd.). Błędy w tych ustawieniach mogą powodować przesyłanie danych w formie otwartej bez świadomości zespołu.

## A03:2021 – Injection

Kategoria *A03:2021 Injection* obejmuje podatności wynikające z **niewłaściwego przetwarzania danych wejściowych**, które mogą zostać użyte do wstrzyknięcia złośliwych instrukcji w różnych komponentach aplikacji. Błędy tego typu prowadzą do sytuacji, w których dane użytkownika są interpretowane jako kod lub część zapytań – zamiast pozostać zwykłymi danymi. Do najczęstszych przykładów należą: wstrzyknięcia SQL (SQL Injection), zdalne wykonanie kodu (Remote Code Execution), wstrzyknięcia poleceń systemowych (Command Injection) oraz manipulacja skryptami po stronie klienta (Cross-Site Scripting).

W kontekście aplikacji Ruby on Rails podatności typu injection mogą wystąpić na różnych poziomach: w bezpośrednich zapytaniach SQL z użyciem interpolowanych zmiennych, przy niekontrolowanym przekazywaniu danych do poleceń systemowych, w widokach HTML renderujących dane wejściowe użytkownika bez ich uprzedniego oczyszczenia, a także w przypadku niewłaściwego zabezpieczenia mechanizmów formularzy i sesji.

### Remote Code Execution (RCE)

#### Opis

Remote Code Execution (RCE) to jedna z najgroźniejszych podatności w aplikacjach webowych, umożliwiająca atakującemu wykonanie dowolnego kodu na serwerze. Skutki takiego ataku są zazwyczaj krytyczne i obejmują przejęcie kontroli nad aplikacją, dostęp do danych lub wykonanie poleceń systemowych.

W Ruby on Rails podatność RCE najczęściej wynika z dynamicznego przetwarzania danych wejściowych, zwłaszcza przy użyciu metod takich jak open, send czy eval, a także podczas deserializacji niezaufanych danych (YAML, Marshal, JSON). Bez odpowiedniej walidacji może to prowadzić do wykonania złośliwego kodu i przejęcia kontroli nad aplikacją.Przykład

#### Przykłady

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

W celu zademonstrowania podatności typu Remote Code Execution (RCE) w aplikacjach Ruby on Rails przygotowano prostą aplikację demonstracyjną, zawierającą celowo wprowadzone błędne mechanizmy przetwarzania danych wejściowych.

Przykład oparty jest na kontrolerze ArticlesController (app/controllers/articles\_controller.rb), który obsługuje różne żądania HTTP i przetwarza parametry przekazywane przez użytkownika.

W wybranych fragmentach kodu celowo pozostawiono podatne konstrukcje umożliwiające wykonanie dowolnego kodu na serwerze po dostarczeniu odpowiednio spreparowanych danych wejściowych.

##### RCE poprzez funkcję open

W demonstracyjnej aplikacji wykorzystano fragment kodu, w którym parametr url przekazywany przez użytkownika trafia bezpośrednio do funkcji open:

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 41

Funkcja open w języku Ruby może działać w kilku trybach: odczytu lokalnych plików (open("/etc/passwd")), pobierania danych z adresów URL (open("http://example.com")), a także – co szczególnie niebezpieczne – wykonania polecenia systemowego, jeśli wartość parametru rozpoczyna się od znaku |.

Przykładowe zapytanie HTTP:



Rysunek 42

spowoduje wykonanie komendy:



Rysunek 43

Po wykonaniu tego zapytania, na serwerze zostanie utworzony plik /tmp/rce1.txt z datą i godziną wykonania komendy. Można to sprawdzić z poziomu powłoki kontenera z działającą aplikacją, poprzez wykonanie komendy:



Rysunek 44

Obraz zawierający tekst, oprogramowanie, Oprogramowanie multimedialne, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 45

Funkcji open nie należy stosować z danymi wejściowymi użytkownika, ponieważ jej działanie zależy od typu przekazanego argumentu i może prowadzić do wykonania polecenia systemowego. W przypadku konieczności pobrania danych z URL-a należy używać dedykowanych bibliotek HTTP i stosować walidację źródła.

##### RCE poprzez funkcję send

Metoda send w języku Ruby umożliwia dynamiczne wywoływanie metod na obiekcie poprzez przekazanie ich nazwy jako ciągu znaków.

Przykładowo, wywołanie user.name można zapisać jako user.send("name"), a jeśli metoda przyjmuje argumenty – jako user.send("greet", "Maciek"). Choć rozwiązanie to daje dużą elastyczność, staje się niebezpieczne, gdy użytkownik ma wpływ na przekazywaną nazwę metody i argumenty. W takiej sytuacji może dojść do wywołania metod prywatnych lub potencjalnie szkodliwych, jak np. eval.

W przygotowanej aplikacji demonstracyjnej zastosowano fragment kodu, który dynamicznie wywołuje metodę na obiekcie @result\_article, na podstawie parametrów wejściowych przekazanych w żądaniu HTTP:

Obraz zawierający tekst, zrzut ekranu, Czcionka, linia

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 46

Przykładowe zapytanie:



Rysunek 47

spowoduje, że aplikacja wykona następującą operację:



Będzie to skutkować uruchomieniem polecenia systemowego i utworzeniem pliku /tmp/rce2.txt z datą i godziną. W języku Ruby zapis polecenia w tzw. backtickach (`...`) powoduje jego wykonanie w systemowej powłoce.

Analogicznie jak w poprzednim przypadku, możemy sami wykonać to zapytanie i sprawdzić wynik działania wewnątrz kontenera z aplikacją:

Obraz zawierający tekst, oprogramowanie, Oprogramowanie multimedialne, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 48

Metody send nie należy używać z danymi wejściowymi od użytkownika. Jeśli jest to konieczne, należy ograniczyć możliwe wywołania do zdefiniowanej listy bezpiecznych metod. Przykładowa ochrona:

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

##### RCE poprzez deserializację

Deserializacja to proces odtwarzania obiektów z danych tekstowych lub binarnych. W Ruby jej nieprawidłowe użycie może prowadzić do wykonania złośliwego kodu na serwerze. Problem dotyczy głównie funkcji takich jak Marshal.load i YAML.unsafe\_load, które odtwarzają dowolne obiekty Ruby – w tym takie, które mogą zawierać polecenia systemowe.

W demonstracyjnej aplikacji podatny fragment wykorzystuje Marshal.load do odczytania danych zakodowanych w formacie Base64. Mimo że w nowszych wersjach Ruby (np. 3.3.5) dodano ograniczenia bezpieczeństwa, podatność nadal występuje w starszych środowiskach. Podobnie działa YAML.unsafe\_load, który może wczytać obiekt z metodą system() zaszytą w strukturze YAML.

Deserializacja to **złożony i trudny temat**, a ze względu na złożoność i wysoki poziom ryzyka, deserializacja danych wejściowych powinna być traktowana jako mechanizm potencjalnie niebezpieczny. Szczegóły zabezpieczania tego procesu wykraczają poza zakres niniejszej pracy.

#### Skutki

Podatność typu Remote Code Execution (RCE) należy do najgroźniejszych luk bezpieczeństwa w aplikacjach webowych. Jej skutki mogą być katastrofalne i obejmować m.in.:

* Przejęcie pełnej kontroli nad serwerem poprzez wykonywanie poleceń systemowych.
* Uzyskanie dostępu do poufnych danych, takich jak hasła, klucze API czy bazy danych.
* Instalację złośliwego oprogramowania (np. backdoora lub ransomware).
* Usunięcie lub modyfikację danych, co może prowadzić do utraty integralności systemu.

#### Zalecenia

##### Walidacja danych wejściowych

Nie należy wykonywać żadnego kodu na podstawie danych pochodzących od użytkownika. Wszystkie parametry wejściowe powinny być walidowane i filtrowane zgodnie z przyjętymi regułami.

##### Unikanie dynamicznego wykonania kodu

Metody takie jak eval, send, public\_send powinny być stosowane wyłącznie w kontrolowanych warunkach. W przypadku konieczności użycia – należy ograniczyć możliwe wartości do zdefiniowanej listy (whitelisty).

##### Bezpieczna deserializacja danych

Należy unikać metod takich jak Marshal.load i YAML.unsafe\_load. Zalecane jest korzystanie z bezpiecznych formatów, np. JSON, oraz metod pozwalających na kontrolę deserializowanych typów (YAML.safe\_load).

##### Ograniczenie uprawnień serwera

Aplikacja powinna działać w odizolowanym środowisku z minimalnymi uprawnieniami systemowymi. Pozwala to ograniczyć skutki ewentualnego ataku.

##### Aktualizacja zależności

Regularna aktualizacja środowiska uruchomieniowego Ruby, frameworka Rails oraz zewnętrznych bibliotek (gemów) pozwala uniknąć znanych podatności i exploitów.

### Cross-Site Scripting (XSS)

#### Opis

Cross-Site Scripting (XSS) to jedna z najczęściej występujących podatności w aplikacjach webowych. Polega na wstrzyknięciu złośliwego kodu JavaScript, który jest następnie wykonywany w kontekście przeglądarki użytkownika. Może to prowadzić do kradzieży sesji, przechwycenia danych, ataków phishingowych lub modyfikacji treści strony.

Wyróżnia się trzy podstawowe typy ataków XSS:

* **Stored XSS** – złośliwy kod zapisywany jest po stronie serwera (np. w bazie danych) i prezentowany innym użytkownikom.
* **Reflected XSS** – złośliwy kod jest przekazywany w parametrze żądania i natychmiast zwracany przez aplikację bez odpowiedniego filtrowania.
* **DOM-based XSS** – podatność występuje po stronie klienta, gdy aplikacja manipuluje drzewem DOM przy użyciu niezaufanych danych wejściowych (np. poprzez innerHTML).

#### Przykład

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

Aplikacja pozwala użytkownikom na dodawanie postów i komentowanie ich bez konieczności logowania. W różnych miejscach aplikacji umieszczono podatne fragmenty kodu, aby zilustrować trzy typy ataków XSS.

Obraz zawierający tekst, zrzut ekranu, numer, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 49

##### Stored XSS

W aplikacji występuje podatność typu *stored XSS*, umożliwiająca zapisanie złośliwego kodu JavaScript w komentarzu, który następnie zostaje wyświetlony innym użytkownikom odwiedzającym stronę. W naszym przypadku, użytkownik o nazwie **Haker** dodał komentarz zawierający kod:



Rysunek 50

Po odświeżeniu strony, złośliwy kod zostaje wykonany automatycznie w przeglądarce – wyświetlany jest alert z komunikatem XSS. Komentarz nie zawiera widocznej treści, ponieważ znacznik <script> nie generuje widocznego tekstu na stronie, a jedynie wykonuje kod.

Obraz zawierający zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne, tekst

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 51

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 52

Domyślnie Rails automatycznie escapuje dane (<%= %>), co zapobiega XSS. Zastosowanie raw powoduje, że dane są interpretowane jako HTML, co umożliwia wykonanie kodu JavaScript.

Rysunek 53

##### Reflected XSS

Podatność typu reflected XSS pojawia się, gdy aplikacja wyświetla dane z parametrów żądania HTTP bez odpowiedniego filtrowania lub escapowania. W przykładowej aplikacji dotyczy to funkcji wyszukiwania postów, gdzie wprowadzony parametr query jest wyświetlany w widoku.

Obraz zawierający zrzut ekranu, tekst, oprogramowanie, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 54

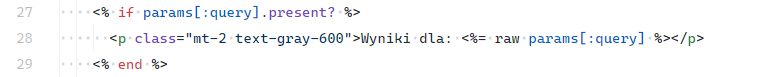
Po wpisaniu złośliwego kodu w pole wyszukiwania, np.: <script>alert('XSS')</script> i kliknięciu przycisku **Szukaj**, kod zostaje natychmiast wykonany w przeglądarce.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 55

Dzieje się tak, ponieważ wyświetlenie parametru query również odbywa się z użyciem raw:



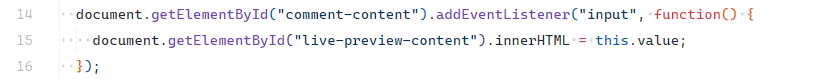
Rysunek 56

Usunięcie raw i pozostanie przy domyślnym escapowaniu (<%= %>) zapobiega podatności.

##### DOM-based XSS

DOM-based XSS to typ ataku, który występuje po stronie klienta – bez udziału serwera – i wynika z niewłaściwego przetwarzania danych wejściowych w przeglądarce. W aplikacji demonstracyjnej zastosowano funkcję podglądu komentarza w czasie rzeczywistym, która aktualizuje zawartość podglądu po każdej zmianie w formularzu.

Kod JavaScript odpowiedzialny za tę funkcję wygląda następująco:



Rysunek 57

Wprowadzenie złośliwego kodu, np.: <img src="x" onerror="alert('XSS!')"> powoduje natychmiastowe wykonanie kodu w przeglądarce – jeszcze przed przesłaniem formularza.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Wynika to z użycia innerHTML, które interpretuje dane jako kod HTML. Aby temu zapobiec, należy zastosować textContent, które traktuje dane jako zwykły tekst:



Rysunek 58

#### Skutki

Ataki typu Cross-Site Scripting mogą prowadzić do poważnych naruszeń bezpieczeństwa użytkowników oraz integralności aplikacji internetowej. W kontekście Ruby on Rails zagrożenia te obejmują:

* Kradzież sesji użytkownika – poprzez przechwycenie ciasteczek sesyjnych, zwłaszcza jeśli aplikacja nie stosuje flagi HttpOnly.
* Phishing i manipulacja interfejsem – złośliwy kod może modyfikować zawartość strony, wyświetlać fałszywe formularze lub komunikaty w celu wyłudzenia danych.
* Zmiana działania aplikacji – JavaScript może zmieniać treści postów, formularzy lub wywoływać działania bez wiedzy użytkownika.

#### Zalecenia

##### Bezpieczne wyświetlanie treści HTML

Rails domyślnie escapuje dane w tagach <%= %>, co chroni przed XSS w większości przypadków. Jeżeli jednak aplikacja dopuszcza wyświetlanie HTML (np. z edytora WYSIWYG lub Markdowna), należy jawnie filtrować dozwolone znaczniki z użyciem metody sanitize, np.:



Rysunek 59

Użycie metod raw lub html\_safe powinno być ograniczone wyłącznie do wcześniej przefiltrowanych danych z zaufanego źródła. Nigdy nie należy stosować ich bezpośrednio na danych wejściowych użytkownika.

##### Bezpieczna manipulacja DOM w JavaScript

Zamiast innerHTML, należy stosować textContent lub inne metody, które nie interpretują wprowadzonego HTML.

##### Wdrożenie Content Security Policy (CSP)

Nagłówki CSP ograniczają możliwość wykonania nieautoryzowanych skryptów. Rails umożliwia konfigurację CSP w pliku config/initializers/content\_security\_policy.rb.

##### Walidacja danych wejściowych

Parametry z formularzy i adresów URL powinny być walidowane pod kątem niedozwolonych znaków lub struktur.

**Zabezpieczenie ciasteczek sesyjnych**

Ciasteczka sesyjne powinny być oznaczone flagą HttpOnly, co uniemożliwia dostęp do nich z poziomu JavaScriptu (np. przez document.cookie). Chroni to użytkowników przed przejęciem sesji w przypadku skutecznego ataku XSS.

### Cross-Site Request Forgery (CSRF)

#### Opis

Cross-Site Request Forgery (CSRF) to atak polegający na wymuszeniu przez złośliwą stronę internetową wykonania nieautoryzowanych operacji w imieniu zalogowanego użytkownika, bez jego wiedzy i zgody.

Mechanizm ten wykorzystuje fakt, że przeglądarka automatycznie dołącza ciasteczka sesyjne do każdego żądania kierowanego do znanej domeny. Jeżeli aplikacja nie posiada odpowiednich zabezpieczeń, możliwe jest wykonanie niezamierzonych działań, takich jak zmiana hasła, wykonanie przelewu czy modyfikacja danych użytkownika.

Framework Ruby on Rails domyślnie chroni aplikacje przed atakami typu CSRF za pomocą mechanizmu tokenów autentyczności (authenticity\_token). Token ten jest generowany automatycznie dla każdego formularza i przypisywany do bieżącej sesji użytkownika.

W przypadku użycia standardowych helperów, takich jak form\_with lub form\_for, token CSRF zostaje automatycznie dodany do formularza jako ukryte pole. Podczas wysyłania formularza, token ten jest przesyłany do serwera, który porównuje go z wartością zapisaną w sesji. Jeżeli token nie zostanie znaleziony, jest niepoprawny lub niezgodny z aktualną sesją użytkownika, framework zgłasza wyjątek InvalidAuthenticityToken, co skutecznie blokuje próbę wykonania żądania.

#### Przykład

Przykładowa aplikacja ilustrująca podatność typu Cross Site Request Forgery znajduje się w katalogu ruby-on-rails-security/cross-site-request-forgery w repozytorium projektowym dla pracy magisterskiej. Po uruchomieniu aplikacji należy przejść pod następujący adres w przeglądarce <http://localhost:3000>.

W celu demonstracji podatności przygotowano aplikację zawierającą dwa kontrolery:

* SafeArticlesController – chroniony domyślnym mechanizmem CSRF.
* UnsafeArticlesController – pozbawiony tej ochrony.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 60

##### Dołączenie tokenu CSRF

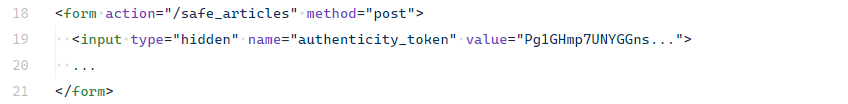
Widok formularza tworzącego artykuł w SafeArticlesController wykorzystuje helper form\_with:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 61

Po wyrenderowaniu formularz zawiera token CSRF jako ukryte pole:



Rysunek 62

Dodatkowo Rails automatycznie dodaje meta-tag z tokenem CSRF w <head> każdej strony HTML (przy użyciu csrf\_meta\_tags), który może być wykorzystywany np. w żądaniach AJAX.



Rysunek 63

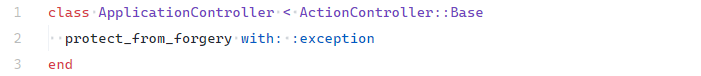
Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 64

##### Domyślna ochrona kontrolerów

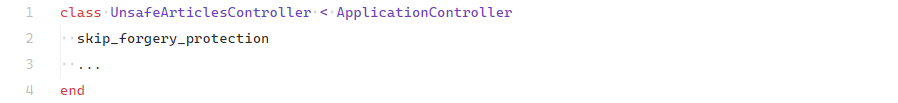
Każdy kontroler dziedziczący po ActionController::Base domyślnie dziedziczy zabezpieczenie CSRF dzięki dyrektywie:



Rysunek 65

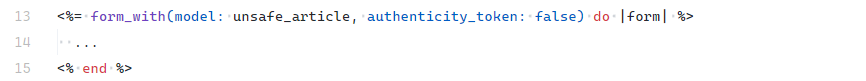
##### Celowe wyłączenie ochrony

W UnsafeArticlesController celowo wyłączono weryfikację tokenu CSRF, stosując metodę skip\_forgery\_protection:



Rysunek 66

Dodatkowo, w widoku formularza tworzono go z wyłączonym generowaniem tokenu. Taki formularz działa prawidłowo, ale **nie chroni przed atakiem CSRF**.



Rysunek 67

##### Symulacja ataku CSRF

Aby zademonstrować podatność, przygotowano osobną stronę HTML udostępnioną jako niezależny serwis działający pod adresem <http://localhost:3001>. Strona ta imituje zachowanie atakującego, który próbuje przesłać formularz do aplikacji ofiary z innej domeny.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Przykładowy formularz na stronie atakującej wygląda następująco:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 68

Po wybraniu opcji utworzenia artykułu, którego kontroler oraz formularz nie posiadają ochrony przed atakami CSRF, aplikacja poprawnie zapisze dane i utworzy nowy wpis – co zostało przedstawione na poniższym zrzucie ekranu:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Strona internetowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 69

W analogiczny sposób wykonanie tego samego ataku na kontroler /safe\_articles poprzez wybranie drugiej opcji na stronie atakującego zakończy się błędem. Rails odrzuca to żądanie, zwracając wyjątek ActionController::InvalidAuthenticityToken.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 70

#### Skutki

Brak ochrony przed CSRF może prowadzić do:

* Nieautoryzowanych działań w imieniu użytkownika – np. zmiany danych, dodania treści lub wykonania operacji bez jego wiedzy.
* Przejęcia konta – zwłaszcza w połączeniu z innymi podatnościami, np. XSS.
* Utrat zaufania i naruszeń prawnych – skutki ataku mogą obejmować manipulację danymi, utratę reputacji i konsekwencje zgodności z regulacjami (np. RODO).

#### Zalecenia

##### Weryfikacja tokenu CSRF

Token CSRF powinien być zawsze sprawdzany po stronie serwera – każda operacja modyfikująca stan aplikacji musi być chroniona mechanizmem porównywania tokenów sesyjnych.

##### Korzystanie z domyślnej ochrony Rails

Nie należy wyłączać protect\_from\_forgery, który domyślnie chroni wszystkie kontrolery HTML. Wyjątki dopuszczalne są wyłącznie w API, gdzie nie korzystamy z ciasteczek do sesji i autoryzacja odbywa się innymi metodami (np. tokeny JWT, OAuth).

##### Używanie helperów Rails

Helpery form\_with i form\_for używane w widokach automatycznie dodają ukryte pole z tokenem CSRF, co eliminuje ryzyko jego pominięcia przy ręcznym tworzeniu formularzy.

##### Obsługa CSRF w zapytaniach AJAX

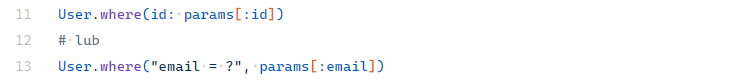
Dla żądań wysyłanych z JavaScriptu (np. fetch lub axios) należy dodać token w nagłówku X-CSRF-Token. Rails umieszcza token automatycznie w znaczniku <meta>, co ułatwia jego pobranie i dodanie do żądań.

### SQL Injection (SQLi)

#### Opis

SQL Injection to to podatność, polegająca na wstrzyknięciu niepożądanego fragmentu zapytania SQL poprzez dane wejściowe użytkownika. Celem ataku może być odczyt, modyfikacja, a nawet usunięcie danych w bazie – z pominięciem logiki autoryzacji i ograniczeń aplikacji.

W kontekście Ruby on Rails operacje na bazie danych realizowane są głównie przy użyciu warstwy ORM Active Record, która – pod warunkiem prawidłowego użycia – zapewnia skuteczną ochronę przed SQL Injection. Bezpieczne konstrukcje to m.in.:



Rysunek 71

Podatność pojawia się jednak w sytuacjach, gdy programista samodzielnie konstruuje fragmenty zapytań SQL, interpolując wartości pochodzące od użytkownika:



Rysunek 72

W tym przypadku wartość parametru jest wstawiana bezpośrednio do zapytania, co pozwala na jego manipulację. Atakujący może np. przesłać 1 OR 1=1--, co spowoduje zwrócenie wszystkich użytkowników, niezależnie od ich identyfikatora. Rails nie wykryje tego jako błąd – zapytanie zostanie wykonane zgodnie z intencją atakującego.

#### Przykład

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

Aplikacja została zaprojektowana tak, aby prezentować różnorodne błędy wynikające z nieprawidłowego wykorzystania zapytań SQL w kontekście Active Record.

Obraz zawierający tekst, zrzut ekranu, numer, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 73

Podstawą aplikacji jest kontroler InjectionsController, który zawiera kilkanaście akcji (np. where, select, join, calculate, delete\_by), z których każda celowo zawiera nieprawidłowe łączenie danych wejściowych z fragmentami SQL. Celem aplikacji jest ukazanie, jak nieprawidłowe wykorzystanie interfejsów ORM może prowadzić do wykonania nieautoryzowanych operacji na bazie danych.

Schemat bazy danych opiera się na dwóch tabelach:

* users (kolumny: name, password, age, admin)
* orders (kolumny: user\_id, total)

Po wejściu na stronę główną aplikacji użytkownik ma możliwość wyboru jednej z akcji kontrolera – każda odpowiada innemu przypadkowi użycia i potencjalnej podatności. Po wybraniu widoku konkretnej akcji prezentowane są następujące elementy:

* **Przycisk "Zresetuj bazę danych"** – umożliwia przywrócenie bazy danych do stanu początkowego z wykorzystaniem predefiniowanych seedów.
* **Kod źródłowy akcji kontrolera** – pokazuje fragment kodu Ruby, w którym znajduje się potencjalna luka bezpieczeństwa.
* **Sekcja "Przykłady użycia"** – zawiera podpowiedzi przykładowych wartości, w tym złośliwych inputów SQL.
* **Formularz** – pozwala na dynamiczne podanie danych, które zostaną wstawione do zapytania SQL.
* **Wygenerowane zapytanie SQL** – wyświetla finalną treść zapytania wygenerowanego przez Active Record, uwzględniającą dane wejściowe użytkownika.
* **Wynik zapytania** – prezentuje rezultat działania zapytania w postaci listy rekordów lub komunikatu o braku wyników.

##### Przykład – akcja where

Na poniższym zrzucie ekran możemy zobaczyć, jak wygląda wynik działania przykładu dla poprawnie podanych parametrów name i password:

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 74

Wygenerowane zapytanie SQL:



Rysunek 75

Wynik zapytania zwrócony przez ActiveRecord:



Rysunek 76

Co się stanie, jeśli zamiast poprawnych danych wprowadzimy złośliwy fragment? Przy wprowadzeniu wartości ' OR 1=1)-- zamiast hasła, zapytanie wygląda następująco:



Rysunek 77

W rezultacie zwracane są wszystkie rekordy z tabeli users, niezależnie od faktycznych danych logowania – ponieważ warunek OR 1=1 jest zawsze prawdziwy:

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 78

Aplikacja w ten sposób jednoznacznie pokazuje, jak łatwo dopuścić do SQL Injection przy nieprawidłowym łączeniu parametrów wejściowych z zapytaniami. Inne Przykłady obejmują różne metody Active Record, m.in. where("..."), select("..."), joins("..."), calculate("..."), i inne – każda z nich może stać się podatna, jeśli zabraknie odpowiedniego zparametryzowania danych.

#### Skutki

Podatność na SQL Injection w aplikacjach Ruby on Rails może prowadzić do:

* **Kradzieży danych** – atakujący może uzyskać dostęp do poufnych informacji, takich jak hasła, dane osobowe czy numery kart płatniczych.
* **Manipulacji i usunięcia danych** – wstrzyknięcie poleceń takich jak DELETE, DROP TABLE czy TRUNCATE może doprowadzić do trwałej utraty danych.
* **Nieautoryzowanego dostępu** – możliwe jest obejście logiki aplikacji i np. nadanie sobie uprawnień administratora.

#### Zalecenia

##### Unikaj dynamicznego dołączania danych do zapytań SQL

Nigdy nie należy wprost doklejać zmiennych, parametrów użytkownika ani innych niekontrolowanych danych do zapytań SQL w postaci łańcuchów znaków. Dotyczy to nie tylko danych pochodzących z formularzy (params), ale również wszelkich źródeł zewnętrznych, takich jak pliki, dane JSON, zmienne środowiskowe czy odpowiedzi z API. Każda informacja spoza kodu źródłowego powinna być traktowana jako potencjalnie niebezpieczna.

##### Zawsze stosuj zapytania parametryzowane

Zamiast konstruowania zapytań poprzez interpolację danych wejściowych:



Rysunek 79

należy stosować bezpieczne zapytania parametryzowane:



Rysunek 80

lub – jeszcze bezpieczniej i czytelniej – z wykorzystaniem haszy:

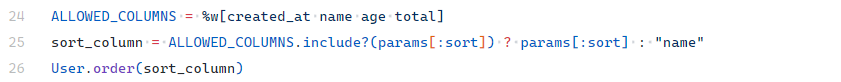


Rysunek 81

W obu przypadkach Active Record automatycznie zabezpiecza zapytanie, przekazując wartości jako tzw. bind parameters, co uniemożliwia wstrzyknięcie dodatkowych fragmentów SQL.

##### Ograniczaj dynamiczne fragmenty zapytań do wartości zdefiniowanych na białej liście

Jeżeli zachodzi konieczność dynamicznego przekazania fragmentu zapytania (np. nazwy kolumny do sortowania), należy upewnić się, że pochodzi on ze ściśle kontrolowanego zbioru dopuszczalnych wartości:



Rysunek 82

## A04:2021 - Insecure Design

Kategoria Insecure Design odnosi się do podatności wynikających z błędnych lub niepełnych założeń projektowych, które skutkują brakiem odpowiednich mechanizmów ochronnych – niezależnie od poprawności implementacji. W odróżnieniu od klasycznych błędów programistycznych, problemy z tej kategorii powstają na etapie projektowania aplikacji lub jej architektury, kiedy nie przewidziano odpowiednich zabezpieczeń przed nadużyciami.

W ramach niniejszej pracy przedstawione zostały trzy wybrane przypadki należące do tej kategorii: podatność typu Open Redirect, brak mechanizmu ograniczenia liczby prób logowania (Login Rate Limiting) oraz problem ReDoS (Regular Expression Denial of Service), czyli możliwość przeciążenia aplikacji przy użyciu specjalnie spreparowanych wyrażeń regularnych

### Open Redirect

#### Opis

Open Redirect to podatność polegająca na braku walidacji adresu URL, na który użytkownik jest przekierowywany. Umożliwia to atakującemu manipulację parametrami żądania i przekierowanie ofiary na zewnętrzną, złośliwą stronę – często w celu phishingu lub obejścia autoryzacji. W Ruby on Rails typowym miejscem wystąpienia tej podatności jest metoda redirect\_to, gdy przyjmuje dane wejściowe bez weryfikacji.

#### Przykład

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

W aplikacji użytkownik loguje się przez standardowy formularz, podając adres e-mail oraz hasło. Po pomyślnym uwierzytelnieniu następuje przekierowanie – domyślnie na stronę główną konta (/dashboard), lub na stronę wskazaną w parametrze redirect\_to. Mechanizm ten ma na celu umożliwienie powrotu do wcześniej odwiedzanej strony po zalogowaniu.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, multimedia

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 83

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, komputer

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 84

Aplikacja nie weryfikuje zawartości parametru redirect\_to, co umożliwia atakującemu przekierowanie użytkownika na zewnętrzną, złośliwą stronę, np. podszywającą się pod oryginalny panel logowania.

##### Scenariusz ataku

Atakujący tworzy złośliwy link i przesyła go użytkownikowi, np. w wiadomości e-mail:



Rysunek 85

Użytkownik trafia na poprawną stronę logowania (/login), co nie wzbudza podejrzeń.

Obraz zawierający tekst, oprogramowanie, Oprogramowanie multimedialne, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 86

Po zalogowaniu aplikacja przekierowuje go jednak na stronę /phishing-login, znajdującą się pod kontrolą atakującego. W rzeczywistym ataku taka strona byłaby wizualnie nieodróżnialna od oryginalnej, by nie wzbudzać podejrzeń. Na potrzeby przykładu została jednak celowo wystylizowana inaczej, aby wyraźnie zaznaczyć, że przekierowanie nastąpiło na fałszywy adres.

Obraz zawierający tekst, oprogramowanie, Oprogramowanie multimedialne, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 87

Widząc ponownie formularz logowania, użytkownik zakłada, że coś poszło nie tak, i wpisuje dane jeszcze raz – tym razem trafiają one bezpośrednio do atakującego. Nie są wysyłane do prawdziwej aplikacji, lecz logowane na stronie phishingowej.

W ten sposób atakujący uzyskuje login i hasło ofiary, co umożliwia mu zalogowanie się na prawdziwe konto, zmianę hasła, zablokowanie użytkownika lub wykonanie nieautoryzowanych operacji, takich jak przelew środków.

#### Skutki

Brak odpowiedniego filtrowania adresów URL podczas przekierowań może prowadzić do poważnych konsekwencji:

* **Phishing i kradzież danych uwierzytelniających** – użytkownik może zostać przekierowany na złośliwą stronę, która podszywa się pod oryginalną aplikację, umożliwiając atakującemu przechwycenie loginu i hasła.
* **Przejęcie sesji i dostęp do konta** – Open Redirect może posłużyć do przechwycenia tokenów sesyjnych, co pozwala na zalogowanie się do aplikacji bez znajomości hasła.

#### Zalecenia

**Waliduj parametry przekierowań względem dozwolonych ścieżek lub domen**

Nie należy używać redirect\_to bez walidacji danych wejściowych – w szczególności wartości parametru redirect\_to. Jeśli aplikacja umożliwia przekierowania na podstawie danych pochodzących od użytkownika, należy zastosować listę dozwolonych ścieżek (whitelist), np.:

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 88

W przypadku gdy przekierowania muszą prowadzić na zewnętrzne domeny (np. partnerzy, systemy płatności), należy sprawdzać hosta adresu URL względem listy zaufanych domen:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 89

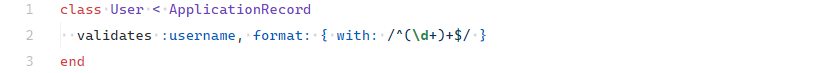
### Regex Denial of Service (ReDoS)

#### Opis

Regex Denial of Service (ReDoS) to atak polegający na wykorzystaniu nieoptymalnych wyrażeń regularnych w celu przeciążenia serwera. Problem pojawia się, gdy wzorzec zawiera rekursywne lub zagnieżdżone grupy, które powodują lawinowy wzrost liczby możliwych dopasowań. Atakujący dostarcza specjalnie spreparowany ciąg znaków, który znacząco wydłuża czas przetwarzania przez parser regexów.

#### Przykład

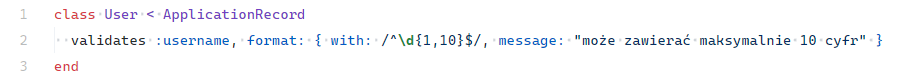
W aplikacji Ruby on Rails podatność typu ReDoS może wystąpić w walidacjach modeli, gdy używane są nieoptymalne wyrażenia regularne. Przykładem jest walidacja pola username z wykorzystaniem wzorca: ^(\d+)+$



Rysunek 90

Wzorzec ten jest podatny na eksplozję kombinatoryczną - przy odpowiednio spreparowanym ciągu, np. 111111111111111111111!, parser regex może wykonywać miliony prób dopasowania, co znacząco obciąża CPU i prowadzi do spowolnienia lub zablokowania aplikacji.

Bezpieczniejszym rozwiązaniem jest zastosowanie wyrażenia ograniczającego długość ciągu:



Rysunek 91

Ograniczenie liczby dopasowań eliminuje ryzyko ReDoS, zachowując jednocześnie funkcjonalność walidacji.

#### Skutki

Skutki podatności ReDoS mogą być poważne, mimo że nie wiążą się bezpośrednio z naruszeniem danych. Ich konsekwencje dotyczą głównie dostępności aplikacji i stabilności działania infrastruktury.

* **Przeciążenie aplikacji** - Nieoptymalne wyrażenia regularne mogą prowadzić do nadmiernego użycia CPU, co skutkuje spowolnieniem działania aplikacji lub jej całkowitym zawieszeniem.
* **Możliwość przeprowadzenia ataku DoS** - Atakujący może wysyłać specjalnie spreparowane dane wejściowe, które wywołują kosztowne obliczeniowo dopasowania, skutkując niedostępnością systemu (Denial of Service).

#### Zalecenia

##### Unikaj dynamicznego tworzenia wyrażeń regularnych na podstawie danych wejściowych

Tworzenie obiektów Regexp z danych dostarczonych przez użytkownika niesie ryzyko wprowadzenia złośliwych wzorców, które mogą prowadzić do przeciążenia aplikacji. Wyrażenia regularne powinny być definiowane statycznie w kodzie, a dane wejściowe – dokładnie walidowane przed ich użyciem.

##### Ogranicz długość i format danych przed ich walidacją

W miarę możliwości, przed zastosowaniem wyrażeń regularnych sprawdzaj długość łańcucha znaków i stosuj limity (np. max\_length).

##### Utrzymuj aktualne wersje Ruby i Rails

Od wersji Ruby 3.2.0 interpreter wprowadza mechanizmy ograniczania czasu wykonywania wyrażeń regularnych oraz optymalizuje ich działanie. Warto korzystać z tych funkcji i dbać o bieżące aktualizacje środowiska.

Przykładowo, w wersjach starszych niż Ruby 3.2.0 podatne wyrażenie mogło zablokować aplikację na ponad 24 sekundy:



Rysunek 92

Od wersji Ruby 3.2.0 to samo wyrażenie kończy się w ułamku sekundy dzięki zastosowanym optymalizacjom:



Rysunek 93

### Login Rate Limiting

#### Opis

Login Rate Limiting to podatność wynikająca z braku ograniczenia liczby prób logowania w aplikacji internetowej. Jej obecność umożliwia atakującemu przeprowadzanie ataków siłowych (brute force) lub automatyczne testowanie danych logowania pochodzących z wycieków (credential stuffing). W efekcie może dojść do nieautoryzowanego dostępu do kont użytkowników. W przypadku aplikacji Ruby on Rails, do wersji 7.2 konieczne było użycie zewnętrznych bibliotek (np. Rack::Attack), natomiast najnowsze wersje frameworka oferują wbudowaną obsługę ograniczeń dzięki mechanizmowi rate\_limit, co znacznie upraszcza wdrażanie ochrony przed tego typu atakami.

#### Przykład

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

W demonstracyjnej aplikacji znajduje się formularz logowania, który nie posiada żadnego ograniczenia liczby prób. Użytkownik – a także potencjalny atakujący – może bez przeszkód wielokrotnie wpisywać dowolne dane logowania.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, multimedia

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 94

Brak mechanizmu rate limiting to nie tylko problem w przypadku logowania. Istnieje szereg innych wrażliwych operacji, które również powinny być objęte ograniczeniami:

* **Resetowanie hasła** – masowe wysyłanie żądań resetujących może prowadzić do spamowania skrzynek e-mail użytkowników lub wywoływać ataki typu account lockout.
* **Weryfikacja kodów OTP (2FA)** – brak limitu prób umożliwia brute-force kodów jednorazowych, co może skutkować przejęciem konta.
* **Rejestracja użytkownika** – brak ograniczeń umożliwia tworzenie tysięcy fałszywych kont, prowadząc do nadużyć lub spamu.
* **Sprawdzanie dostępności nazwy użytkownika** – umożliwia enumerację kont, co sprzyja atakom typu credential stuffing.
* **Zadania zasobożerne** – niekontrolowany dostęp do kosztownych operacji (np. eksportów, generowania raportów) może prowadzić do nadmiernego zużycia zasobów i skutecznego ataku DoS.

#### Skutki

Brak mechanizmu ograniczającego liczbę żądań (rate limiting) w aplikacji webowej może prowadzić do poważnych zagrożeń zarówno dla użytkowników, jak i całego systemu. W szczególności:

* **Nieautoryzowany dostęp do konta** – ataki typu brute-force i credential stuffing umożliwiają systematyczne sprawdzanie haseł (lub skradzionych danych logowania), aż do skutecznego zalogowania się na konto ofiary.
* **Obejście mechanizmów bezpieczeństwa** – brak limitów prób dla operacji takich jak reset hasła czy weryfikacja kodów 2FA może umożliwić atakującemu przejęcie konta mimo dodatkowych zabezpieczeń.
* **Nadużycia systemu i ataki DoS** – nadmierna liczba żądań (np. prób rejestracji, resetów haseł czy kosztownych operacji backendowych) może obciążyć serwer i spowodować jego niedostępność (Denial of Service).

#### Zalecenia

##### Rails 7.2 i nowsze

Od wersji 7.2 Ruby on Rails oferuje natywny mechanizm ograniczania liczby żądań (rate\_limit), który pozwala bezpośrednio w kontrolerach definiować limity dla wybranych akcji.

Przykładowa implementacja w kontrolerze logowania:

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 95

Opis działania:

* to: 5 – maksymalnie 5 żądań (np. prób logowania),
* within: 1.minute – w ciągu jednej minuty,
* with: – po przekroczeniu limitu użytkownik zostaje przekierowany na specjalną stronę informacyjną,
* only: – ograniczenie dotyczy tylko konkretnej akcji (np. create).

Jeśli użytkownik przekroczy liczbę dopuszczalnych prób logowania, zostanie automatycznie przekierowany na zdefiniowaną stronę, np. rate\_limited\_path, z informacją o blokadzie czasowej, tak jak to widać na rysunku poniżej:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, komputer

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 96

##### Rails starsze niż 7.2

Dla starszych wersji Rails zalecanym rozwiązaniem jest wykorzystanie biblioteki Rack Attack, która działa jako middleware dla każdego zapytania w aplikacji i pozwala wdrożyć mechanizm rate limiting niezależnie od wersji frameworka.

Przykład konfiguracji ograniczenia liczby prób logowania na podstawie adresu IP:

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 97

Rack Attack umożliwia również ochronę innych punktów aplikacji narażonych na nadużycia, takich jak: resetowanie hasła, weryfikacja kodów 2FA, rejestracja konta, żądania do API itp.

## A05:2021 - Security Misconfiguration

Kategoria Security Misconfiguration odnosi się do błędów w konfiguracji środowiska aplikacji, które mogą prowadzić do poważnych naruszeń bezpieczeństwa. Obejmuje to zarówno domyślne ustawienia frameworka, niepotrzebnie otwarte punkty końcowe (endpointy), nieprawidłowo skonfigurowane nagłówki HTTP, jak i brak ograniczeń dostępu do zasobów systemowych.

W kontekście aplikacji webowych opartych na Ruby on Rails tego typu podatności mogą pojawić się m.in. w niewłaściwej konfiguracji ciasteczek sesyjnych lub tokenów (np. brak flag HttpOnly i Secure), niezabezpieczonym przetwarzaniu danych XML (np. podatność XXE), braku wymuszenia HTTPS (brak HSTS) czy uruchomieniu aplikacji z aktywnym trybem debugowania w środowisku produkcyjnym.

### Token / Cookie Misconfiguration

#### Opis

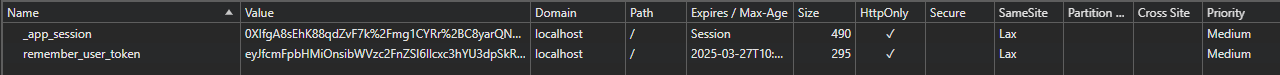
Niewłaściwa konfiguracja tokenów i ciasteczek (cookies) może prowadzić do szeregu zagrożeń bezpieczeństwa, takich jak przejęcie sesji użytkownika, ataki CSRF i XSS, czy nieautoryzowany dostęp do danych. Błędy takie obejmują m.in. brak ustawienia flag Secure, HttpOnly lub SameSite, zbyt długi czas życia ciasteczka, czy przechowywanie wrażliwych danych bez odpowiedniego szyfrowania.

#### Przykład

W aplikacjach webowych sesja to mechanizm umożliwiający przechowywanie danych użytkownika pomiędzy kolejnymi żądaniami HTTP. W Ruby on Rails do obsługi sesji wykorzystuje się mechanizmy oparte na ciasteczkach (cookies) lub przechowywane po stronie serwera. Wybór konkretnego rozwiązania wpływa bezpośrednio na poziom bezpieczeństwa aplikacji.

Rails oferuje kilka metod przechowywania sesji:

* CookieStore (domyślny) – zapisuje zaszyfrowaną zawartość sesji po stronie klienta, w ciasteczku.
* CacheStore, MemCacheStore, ActiveRecordStore – przechowują sesję po stronie serwera, a w ciasteczku zapisany jest jedynie identyfikator sesji.

Aby zilustrować działanie domyślnego mechanizmu, rozważmy przykładową aplikację korzystającą z biblioteki Devise i domyślnej konfiguracji sesji. Po zalogowaniu użytkownika można zaobserwować zawartość ciasteczka sesyjnego w narzędziach deweloperskich przeglądarki

Rysunek 98

Na jego podstawie można zauważyć domyślne ustawienia:

* Aplikacja korzysta z CookieStore, czyli cała sesja przechowywana jest po stronie klienta w ciasteczku.
* Nazwa ciasteczka to zwykle \_app\_session, gdzie app to prefiks wynikający z nazwy aplikacji. Zawartość ciasteczka jest podpisana i zaszyfrowana.
* Flaga HttpOnly jest domyślnie włączona, SameSite ustawione na Lax, natomiast brakuje flagi Secure.

W Ruby on Rails ciasteczka są domyślnie:

* **Podpisane** (signed) – dzięki temu nie można zmienić ich treści bez unieważnienia podpisu (integralność).
* **Zaszyfrowane** (encrypted) – użytkownik nie jest w stanie odczytać danych przechowywanych w ciasteczku (poufność).

#### Skutki

Przykład błędnie skonfigurowanego ciasteczka:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 99

Niepoprawnie ustawione flagi ciasteczek mogą poważnie narazić aplikację na ataki:

* **Przechwycenie sesji** (Session Hijacking) - Brak flagi Secure umożliwia przesyłanie ciasteczek przez HTTP, co w sieci niezabezpieczonej (np. Wi-Fi) pozwala atakującemu na ich przechwycenie i przejęcie sesji użytkownika.
* **Ataki XSS i dostęp do ciasteczek przez JavaScript** - Brak HttpOnly pozwala na odczytanie ciasteczek przez złośliwe skrypty osadzone w stronie (np. document.cookie), co może prowadzić do kradzieży ciasteczek.
* **CSRF** - Przy ustawieniu SameSite: :none ciasteczka będą przesyłane do aplikacji także w kontekście zewnętrznych stron, co umożliwia wykonanie nieautoryzowanych żądań.
* **Ujawnienie danych przechowywanych w ciasteczku** - Jeżeli aplikacja korzysta z CookieStore, a dane nie są odpowiednio zaszyfrowane, użytkownik może je podejrzeć lub nawet zmodyfikować.

#### Zalecenia

##### Jawna i poprawna konfiguracja ciasteczek lub sesji

W celu zapewnienia bezpieczeństwa sesji użytkownika, ciasteczka HTTP mogą być konfigurowane za pomocą następujących flag:

* Expires / Max-Age – określa czas życia ciasteczka. Brak tej opcji oznacza, że ciasteczko zostanie usunięte po zamknięciu przeglądarki.
* Domain – definiuje, dla jakiej domeny ciasteczko jest dostępne. Domyślnie ograniczone do bieżącej domeny.
* Path – ogranicza ciasteczko do wybranej ścieżki URL, np. /admin sprawi, że będzie ono dostępne tylko pod tą ścieżką.
* SameSite – ogranicza przesyłanie ciasteczek w żądaniach cross-site (między różnymi domenami), zapobiegając atakom CSRF:
  + Strict – ciasteczko wysyłane tylko w obrębie tej samej domeny.
  + Lax – (domyślna wartość w Rails) – ciasteczko wysyłane m.in. przy żądaniach GET (np. po kliknięciu w link).
  + None – ciasteczko wysyłane zawsze, również między domenami, ale wymaga ustawienia Secure.
* Secure – ciasteczko przesyłane wyłącznie przez HTTPS, chroni przed przechwyceniem w sieciach niezabezpieczonych.
* HttpOnly – blokuje dostęp do ciasteczka z poziomu JavaScriptu (np. document.cookie), co zabezpiecza przed kradzieżą sesji przez XSS.

W środowiskach produkcyjnych zaleca się ręczne ustawienie parametrów sesji w plikach config/environments/\*.rb, zamiast polegania na domyślnych wartościach. Przykład zalecanej konfiguracji w pliku production.rb:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 100

##### Wybór odpowiedniego mechanizmu przechowywania sesji

Wybór między CookieStore, a mechanizmami serwerowymi (CacheStore, ActiveRecordStore) zależy od wymagań aplikacji.

Zalety i ograniczenia CookieStore:

* Prosty i szybki – brak potrzeby konfiguracji zewnętrznego systemu.
* Dane sesyjne przechowywane po stronie klienta (w przeglądarce).
* Ograniczenie rozmiaru do ~4KB.
* Brak możliwości unieważnienia jednej sesji (wszystkie dzielą ten sam klucz).

Zalety ActiveRecordStore / CacheStore:

* Możliwość centralnego zarządzania sesjami. - dane przechowywane są po stronie serwera.
* Możliwość unieważnienia konkretnej sesji (np. po zmianie hasła).
* Brak ograniczenia rozmiaru.
* Wymaga dodatkowej konfiguracji i dbania o czyszczenie nieaktywnych sesji.

W mniejszych aplikacjach warto rozpocząć od CookieStore. W przypadku większej skali lub potrzeby precyzyjnego zarządzania sesjami warto przejść na ActiveRecordStore lub CacheStore.

Na koniec warto pamiętać, że zmienna środowiskowa SECRET\_KEY\_BASE w konfiguracji Railsów stanowi podstawę do szyfrowania i podpisywania ciasteczek dla CookieStore. Utrata poufności tego klucza umożliwia odszyfrowanie sesji i potencjalne ich modyfikowanie.

### XML XXE

#### Opis

XML External Entity (XXE) to typ ataku wykorzystujący lukę w konfiguracji parserów XML. Polega na tym, że parser umożliwia przetwarzanie tzw. zewnętrznych encji (external entities), co pozwala na automatyczne ładowanie zasobów z zewnętrznych źródeł – takich jak lokalne pliki systemowe czy adresy URL.

Choć podatność ta może wydawać się niszowa, należy pamiętać, że wiele popularnych formatów dokumentów – takich jak Microsoft Word (DOCX) czy Excel (XLSX) – opiera się wewnętrznie na strukturze XML. W związku z tym XXE stanowi realne zagrożenie m.in. w systemach finansowych, korporacyjnych i wszędzie tam, gdzie użytkownicy przesyłają załączniki lub importują dane z plików.

#### Przykład

Rozważmy prosty, nieszkodliwy dokument XML przesyłany przez użytkownika:



Rysunek 101

W scenariuszu ataku, osoba nieuprawniona może zmodyfikować treść pliku i dodać do niego deklarację zewnętrznej encji (DOCTYPE), wskazującej na plik znajdujący się na serwerze:

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 102

Jeśli parser XML po stronie aplikacji nie ma zablokowanego przetwarzania zewnętrznych encji, do zmiennej username zostanie wczytana zawartość pliku secrets.yml, znajdującego się lokalnie na serwerze. W rezultacie aplikacja może przypadkowo ujawnić dane konfiguracyjne, takie jak klucze API, hasła czy sekrety sesji.

#### Skutki

Ataki XXE mogą mieć poważne konsekwencje dla aplikacji:

* **Ujawnienie poufnych danych** – parser może załadować zawartość lokalnych plików (np. secrets.yml, .env), ujawniając hasła, klucze API czy konfiguracje.
* **Ataki typu Denial-of-Service** (DoS) – poprzez zewnętrzne encje odwołujące się do dużych lub nieskończonych zasobów (np. XML bomb), możliwe jest zablokowanie działania parsera lub całej aplikacji.
* **Atak SSRF** (Server-Side Request Forgery) – XXE może zostać wykorzystane do wykonywania zapytań HTTP z serwera do wewnętrznych zasobów (np. http://localhost, 169.254.169.254), co może prowadzić do eskalacji uprawnień, skanowania portów lub uzyskania dostępu do metadanych instancji w środowiskach chmurowych.

#### Zalecenia

##### Unikaj parsowania XML, jeśli to nie jest konieczne

Najprostszym sposobem zabezpieczenia się przed XXE jest całkowita rezygnacja z przetwarzania danych w formacie XML, jeśli nie jest to niezbędne. Format JSON jest bezpieczniejszą alternatywą w większości zastosowań.

##### Korzystaj z bezpiecznego parsera XML

Rails domyślnie wykorzystuje bibliotekę **REXML**, która nie obsługuje zewnętrznych encji i jest odporna na XXE. Jeśli nie istnieje wyraźna potrzeba zmiany parsera, zaleca się pozostanie przy domyślnym rozwiązaniu.

##### Wyłącz obsługę zewnętrznych encji w niestandardowych parserach

Jeśli aplikacja musi korzystać z innego parsera (np. LibXML), należy jawnie wyłączyć możliwość przetwarzania zewnętrznych encji. Przykład konfiguracji

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 103

##### Waliduj zawartość dokumentów XML przed ich przetworzeniem

W przypadku ręcznego przetwarzania danych wejściowych warto dodatkowo sprawdzać, czy przesłany dokument nie zawiera niebezpiecznych deklaracji typu <!ENTITY>:



Rysunek 104

### TLS Force / HSTS

#### Opis

Brak szyfrowania połączenia między przeglądarką a serwerem naraża użytkownika na przechwycenie poufnych danych, takich jak hasła czy identyfikatory sesji. Aplikacja powinna zawsze korzystać z HTTPS, aby zapewnić poufność i integralność przesyłanych informacji. Dodatkowo, mechanizmy takie jak HSTS mogą wymusić bezpieczne połączenia i utrudnić ataki typu SSL stripping.

#### Skutki

Brak odpowiedniego zabezpieczenia transmisji danych między klientem a serwerem może prowadzić do poważnych konsekwencji bezpieczeństwa. Poniżej przedstawiono najczęstsze błędy konfiguracyjne wraz z ich skutkami:

* **Brak wymuszania HTTPS** – jeśli aplikacja akceptuje połączenia HTTP, dane mogą zostać przechwycone przez atakującego w ataku typu Man-in-the-Middle (MITM). Narzędzia takie jak Wireshark pozwalają na podsłuchanie niezabezpieczonego ruchu i wyciągnięcie haseł, tokenów czy danych osobowych. W skrajnych przypadkach możliwe jest przejęcie konta użytkownika. Brak HSTS zwiększa podatność na ataki typu SSL Stripping.
* **Przesyłanie danych w URL** – tokeny sesyjne lub inne wrażliwe informacje przekazywane w parametrach adresu URL mogą zostać przypadkowo zapisane w logach serwera lub historii przeglądarki, co prowadzi do ich wycieku.
* **Brak flag Secure i HttpOnly dla ciasteczek** – umożliwia przesyłanie ciasteczek przez nieszyfrowane połączenia oraz ich odczytanie z poziomu JavaScript, co otwiera drogę do ataków XSS i kradzieży sesji (session hijacking).
* **Brak szyfrowania danych w treści** żądania – przesyłanie np. numerów kart kredytowych w postaci jawnej naraża użytkowników na kradzież danych finansowych.
* **Błędna konfiguracja infrastruktury (proxy, load balancerów)** – nawet jeśli aplikacja wymusza HTTPS, nieprawidłowe przekazywanie żądań wewnątrz infrastruktury może spowodować, że dane przechodzą niezabezpieczoną ścieżką.

#### Zalecenia

##### Wymuś HTTPS w aplikacji Ruby on Rails

Najważniejszym krokiem jest zapewnienie, że cała komunikacja między przeglądarką a aplikacją odbywa się wyłącznie przez HTTPS. W Rails służy do tego opcja config.force\_ssl, którą definiuje się w pliku konfiguracyjnym odpowiednim dla danego środowiska:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 105

Zastosowanie tej opcji zależy od sposobu wdrożenia aplikacji:

* W przypadku aplikacji działającej samodzielnie (np. na VPS, bezpośrednio na serwerze) force\_ssl = true to najprostsze i w pełni automatyczne rozwiązanie.
  + Włącza ono przekierowania z HTTP na HTTPS, ustawia flagę Secure dla ciasteczek oraz dodaje nagłówek HSTS w odpowiedzi serwera.
* W środowiskach z reverse proxy lub load balancerem (np. Kubernetes + Ingress) – aplikacja Rails może nie widzieć, że połączenie było szyfrowane, ponieważ ruch HTTPS zakończy się wcześniej, na warstwie proxy. W takim przypadku:
  + przekierowania i HSTS należy wymusić na poziomie proxy (np. Ingress NGINX, Traefik),
  + w samej aplikacji należy ręcznie skonfigurować ustawienia ciasteczek (secure: true, httponly: true, same\_site: :strict),
  + trzeba upewnić się, że aplikacja prawidłowo interpretuje nagłówek   
    X-Forwarded-Proto.

##### Usunięcie wrażliwych danych z URL

Nie należy przesyłać poufnych informacji (np. tokenów, haseł resetujących, identyfikatorów sesji) jako parametrów w adresach URL.

Adresy URL mogą zostać przypadkowo ujawnione, ponieważ są często logowane przez serwer, zapisywane w historii przeglądarki oraz przesyłane w nagłówkach Referer do zewnętrznych serwisów.

Zamiast tego, wrażliwe dane powinny być przekazywane bezpieczniej – np. w nagłówkach HTTP lub w treści żądania (body):

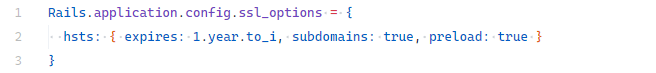
##### Włączenie mechanizmu HSTS

HTTP Strict Transport Security (HSTS) to mechanizm zabezpieczający, który wymusza użycie HTTPS – nawet jeśli użytkownik ręcznie wpisze adres z prefiksem http://. Chroni to przed atakami typu Man-in-the-Middle oraz SSL Stripping. Przykład nagłówka HSTS:



Rysunek 106

W Ruby on Rails nagłówek HSTS jest domyślnie dodawany automatycznie po włączeniu opcji config.force\_ssl. Jeśli potrzebna jest pełniejsza kontrola nad konfiguracją HSTS, można ją dostosować w pliku konfiguracyjnym środowiska:



Rysunek 107

Dobre praktyki przy wdrażaniu HSTS obejmują rozpoczęcie od niskiego parametru max-age na etapie testów, upewnienie się, że wszystkie zasoby i subdomeny obsługują HTTPS oraz regularne monitorowanie ważności certyfikatów TLS.

### Debug Mode

W Ruby on Rails nie istnieje dedykowany tryb "debug mode", jak ma to miejsce w innych frameworkach, takich jak Symfony czy Django. Zamiast tego Rails wykorzystuje różne środowiska pracy (environments), które wpływają na sposób działania aplikacji oraz poziom ujawnianych informacji diagnostycznych.

Domyślnie Rails obsługuje trzy główne środowiska:

* **development** – przeznaczone do programowania, z obszernymi komunikatami błędów oraz narzędziami debugowania,
* **test** – używane do automatycznych testów, bez pełnej obsługi debugowania,
* **production** – zoptymalizowane pod kątem bezpieczeństwa i wydajności, bez szczegółowych komunikatów błędów.

Ponieważ Rails nie posiada dedykowanego trybu debugowania, głównym zagrożeniem jest przypadkowe ujawnienie wrażliwych informacji w wyniku nieprawidłowej konfiguracji środowiska lub pozostawienia narzędzi diagnostycznych aktywnych na produkcji.

Warto również zwrócić uwagę na gemy pomocnicze, takie jak better\_errors, rack-mini-profiler czy bullet, które są niezwykle przydatne podczas developmentu. Dodają one specjalne mechanizmy diagnostyczne bezpośrednio do interfejsu aplikacji – np. wyświetlanie szczegółowych błędów, zapytań SQL czy ostrzeżeń o problematycznym N+1 query. W praktyce stanowią one pewnego rodzaju interaktywny tryb debugowania w przeglądarce.

Tego typu narzędzia nie powinny być aktywne na produkcji, ponieważ mogą ujawniać wrażliwe informacje o wewnętrznej strukturze aplikacji. Aby temu zapobiec, należy upewnić się, że zostały umieszczone w odpowiedniej grupie w pliku Gemfile, np.:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 108

## A09:2021 - Security Logging and Monitoring Failures

Security Logging and Monitoring Failures to kategoria podatności związana z brakiem odpowiedniego rejestrowania i monitorowania zdarzeń w aplikacji. Gdy system nie loguje kluczowych operacji – takich jak próby logowania, błędy autoryzacji, modyfikacje danych czy nieudane próby dostępu – organizacja traci możliwość wykrywania ataków i szybkiego reagowania na incydenty bezpieczeństwa.

### Enable Logging

#### Opis

Logowanie w aplikacjach webowych jest kluczowym elementem monitorowania bezpieczeństwa i wykrywania incydentów. W kontekście Ruby on Rails framework domyślnie oferuje mechanizmy logowania, jednak ich odpowiednia konfiguracja oraz świadome wykorzystanie są niezbędne, aby zapewnić właściwy poziom bezpieczeństwa i diagnostyki.

Nieprawidłowe ustawienia mogą prowadzić do pomijania istotnych zdarzeń lub przeciwnie – nadmiernego logowania, które utrudnia analizę oraz może stanowić zagrożenie bezpieczeństwa, na przykład poprzez ujawnienie danych wrażliwych.

#### Przykład

##### Domyślne logowanie i środowiska w Ruby on Rails

Rails domyślnie tworzy logi osobno dla każdego środowiska aplikacji:

* **development** (log/development.log) - poziom logowania: :debug
* **production** (log/production.log) - poziom logowania: :info
* **test** (log/test.log) - poziom logowania: :warn

To, które środowisko jest aktywne, określa zmienna środowiskowa RAILS\_ENV – na jej podstawie Rails ładuje odpowiednią konfigurację i loguje dane do właściwego pliku.

##### Poziomy logowania w Rails

Rails korzysta z poziomów logowania opartych na bibliotece Logger, które można skonfigurować globalnie (w config/application.rb) lub osobno dla każdego środowiska (config/environments/\*.rb) za pomocą opcji config.log\_level:

* :debug – bardzo szczegółowe logi pomocne w fazie developmentu (np. zapytania SQL, parametry żądań, dane sesji).
* :info – ogólne informacje o działaniu aplikacji, np. przetwarzane akcje, czas odpowiedzi.
* :warn – ostrzeżenia o potencjalnie niebezpiecznych lub nieoptymalnych sytuacjach.
* :error – błędy, które wpłynęły na działanie aplikacji, ale jej nie zatrzymały.
* :fatal – krytyczne błędy powodujące zatrzymanie działania aplikacji.

#### Skutki

Brak odpowiedniego logowania w aplikacji może prowadzić do poważnych problemów w zakresie bezpieczeństwa i utrzymania systemu:

* **Trudności w wykrywaniu ataków** – brak zapisów o podejrzanych logowaniach, manipulacjach w danych czy błędnych próbach uwierzytelnienia utrudnia szybką reakcję na incydenty.
* **Utrudniona diagnostyka błędów** – w przypadku awarii aplikacji lub zgłoszeń użytkowników brak logów uniemożliwia skuteczne debugowanie i analizę źródła problemu.
* **Brak możliwości przeprowadzenia analizy incydentu** – jeśli dojdzie do naruszenia bezpieczeństwa, brak historii zdarzeń uniemożliwia ustalenie przebiegu ataku i jego skutków.

#### Zalecenia

##### Dostosuj poziom logowania do środowiska

W Rails logowanie jest domyślnie włączone, ale poziom szczegółowości powinien być odpowiednio dopasowany: Przykład konfiguracji dla środowiska produkcyjnego w config/environments/production.rb:



Rysunek 109

##### Zadbaj o przechowywanie i rotację logów

Na serwerach VPS logi zapisywane są do plików, np. log/production.log. Aby uniknąć ich nadmiernego rozrastania się, stosuj rotację:



Rysunek 110

Powyższa konfiguracja oznacza, że aplikacja będzie przechowywać maksymalnie 10 plików logów, z których każdy może mieć do 100 megabajtów rozmiaru.

**Logowanie w środowiskach kontenerowych**

W środowiskach kontenerowych (np. Docker, Kubernetes) aplikacje są stateless, a dane zapisane wewnątrz kontenera mogą zostać utracone po restarcie. Dlatego logi nie powinny trafiać do plików lokalnych, lecz na STDOUT – skąd mogą być przechwytywane przez system klastra. W Rails taka konfiguracja może wyglądać następująco:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 111

Dzięki temu logi będą dostępne dla systemów zbierających dane, takich jak Fluentd czy rozwiązania typu ELK Stack (Elasticsearch, Logstash, Kibana) lub DataDog, umożliwiające centralną analizę i wizualizację logów w czasie rzeczywistym.

##### Logowanie kluczowych zdarzeń w systemie

Rejestrowanie istotnych operacji, takich jak logowania, zmiany uprawnień czy modyfikacje danych, jest niezbędne do zapewnienia bezpieczeństwa, analizy incydentów i audytu. Szczególnie warto zapisywać zarówno udane, jak i nieudane próby logowania – pomaga to wykrywać ataki brute-force oraz nieautoryzowane dostępy.

W aplikacjach korzystających z Devise, logowanie prób logowania można dodać np. w kontrolerze sesji:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 112

##### Logowanie zmian w modelach danych

Do śledzenia zmian w kluczowych modelach (np. użytkownicy, zamówienia) świetnie nadaje się gem PaperTrail. Rejestruje on wszystkie operacje wykonywane na rekordach za pośrednictwem ActiveRecord - tworzenie, edycję i usuwanie - zapewniając pełny audyt historii zmian i możliwość przywracania poprzednich wersji danych.

Należy jednak pamiętać, że zmiany wprowadzone bezpośrednio w bazie danych (z pominięciem Railsów) nie zostaną zapisane. Poniżej przykład użycia w modelu User:



Rysunek 113

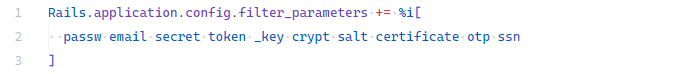
### Logging Sensitive Information

#### Opis

Niewłaściwe logowanie danych wrażliwych, takich jak hasła, tokeny API czy numery kart kredytowych, może prowadzić do ich niezamierzonego ujawnienia i poważnych naruszeń bezpieczeństwa. W Ruby on Rails istnieją mechanizmy zapobiegające temu ryzyku, takie jak filter\_parameters i filter\_attributes, które automatycznie ukrywają określone pola w logach. Jednak brak ich konfiguracji, zbyt ogólne filtrowanie lub ręczne logowanie danych bez uwzględnienia prywatności może sprawić, że poufne informacje trafią do plików logów lub zewnętrznych systemów monitorujących, stając się łatwym celem dla atakujących lub nieuprawnionych użytkowników.

#### Przykłady

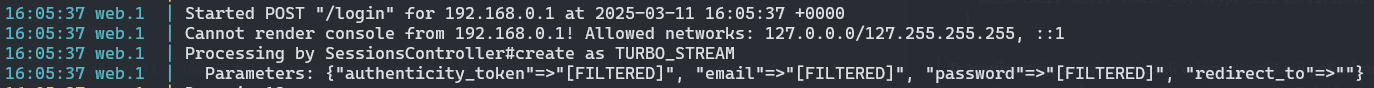
W aplikacji Ruby on Rails domyślnie skonfigurowany jest mechanizm filtracji wrażliwych danych, które mogłyby trafić do logów. W pliku config/initializers/filter\_parameters.rb znajduje się lista słów kluczowych, według których Rails automatycznie maskuje dane:



Rysunek 114

Jeśli którykolwiek z przekazanych parametrów w żądaniu zawiera chociaż część zdefiniowanego słowa w nazwie, jego wartość zostanie zastąpiona etykietą [FILTERED].

Na poniższym zrzucie ekranu widać efekt działania tej konfiguracji — logi kontrolera nie zawierają rzeczywistego hasła, tylko jego zamaskowaną wersję:



Rysunek 115

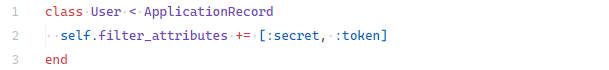
Te same zasady filtracji obowiązują również podczas pracy w konsoli railsowej (rails console). Nawet tam, parametry uznane za wrażliwe nie będą wyświetlane w jawnej postaci:

Obraz zawierający tekst, oprogramowanie, Oprogramowanie multimedialne, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

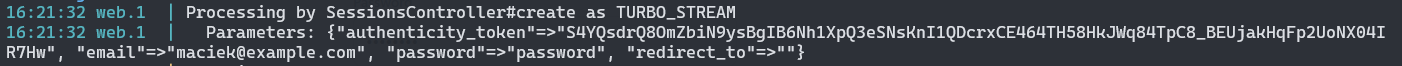
Rysunek 116

Dodatkowo można określić, które atrybuty mają być ukrywane również na poziomie modeli ActiveRecord, a nie globalnej konfiguracji, wykorzystując metodę self.filter\_attributes. Przykład w modelu User:



Rysunek 117

Jednak jeśli te mechanizmy zostaną wyłączone lub pominięte — np. przez nieuwzględnienie danych w konfiguracji filtrów — wrażliwe informacje trafią do logów w pełnej postaci. Poniżej przykład sytuacji, gdy ochrona została celowo wyłączona. Parametry żądania, w tym hasło użytkownika, zostały zapisane w logach bez żadnego filtrowania:



Rysunek 118

#### Skutki

Pliki logów są często przechowywane w formie tekstowej i dostępne lokalnie na serwerze lub przesyłane do zewnętrznych usług monitorujących. W przypadku ich przejęcia lub niewłaściwej konfiguracji mogą zostać łatwo przeszukane przez atakujących lub osoby nieupoważnione.

Dodatkowo, w środowiskach korporacyjnych, nieprzemyślane logowanie może naruszać polityki prywatności lub przepisy o ochronie danych osobowych (np. RODO).

#### Zalecenia

##### Korzystaj z filter\_parameters oraz filter\_attributes

Rails domyślnie filtruje dane takie jak hasła czy tokeny. Lista słów kluczowych powinna być aktualizowana wraz z rozwojem aplikacji, aby objąć wszystkie nowe pola zawierające dane poufne.

##### Unikaj logowania pełnych struktur danych

Nie loguj pełnych obiektów ActiveRecord, całych żądań HTTP ani danych wejściowych użytkownika bez filtracji. W logach powinny znaleźć się wyłącznie niezbędne informacje diagnostyczne: błędy, wyjątki, działania systemowe.

##### Zachowaj czujność w przypadku zewnętrznych usług

Jeśli logi są przesyłane do zewnętrznych systemów monitorowania, należy upewnić się, że nie zawierają danych, które nie powinny opuszczać środowiska aplikacji.

### No Log to User

#### Opis

Bezpieczeństwo aplikacji webowych wymaga, aby szczegóły błędów i wyjątków nie były ujawniane użytkownikom końcowym. Wycieki informacji o wyjątkach, ścieżkach systemowych czy strukturze aplikacji mogą pomóc potencjalnym atakującym w znalezieniu luk w zabezpieczeniach. Ruby on Rails domyślnie wyświetla pełne komunikaty błędów w środowisku deweloperskim, ale na produkcji powinno to być wyłączone.

#### Przykład

W środowisku deweloperskim Ruby on Rails domyślnie prezentuje szczegółowe strony błędów, zawierające pełen komunikat wyjątku, stack trace oraz podgląd kodu w miejscu, w którym wystąpił problem.

Jest to pomocne podczas programowania, ale stanowi poważne zagrożenie bezpieczeństwa w środowisku produkcyjnym – ujawniając wewnętrzne szczegóły aplikacji, w tym strukturę kontrolerów, ścieżki plików czy fragmenty logiki biznesowej.

Zachowanie to jest kontrolowane przez ustawienie w pliku config/environments/development.rb:



Rysunek 119

W takim przypadku, gdy aplikacja napotka błąd, użytkownik zobaczy pełny raport błędu z informacjami diagnostycznymi, tak jak na przykład na poniższym rysunku:

Obraz zawierający tekst, zrzut ekranu, Czcionka, Strona internetowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 120

#### Zalecenia

##### Nie ujawniaj szczegółów błędów na produkcji

W środowisku produkcyjnym (config/environments/production.rb) należy upewnić się, że opcja consider\_all\_requests\_local jest ustawiona na false – co jest domyślnym zachowaniem w Rails. Dzięki temu użytkownik końcowy nie ma dostępu do wewnętrznych szczegółów aplikacji jak np. stack trace itp.



Rysunek 121

W przypadku błędu na produkcji użytkownik zobaczy jedynie ogólną stronę błędu z kodem HTTP 500, bez żadnych poufnych informacji:

Obraz zawierający tekst, zrzut ekranu, Czcionka, wizytówka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 122

##### Logowanie wyjątków do systemu zewnętrznego (np. Sentry)

Samo zapisywanie błędów w logach produkcyjnych może nie wystarczyć. Narzędzia takie jak Sentry umożliwiają monitorowanie wyjątków w czasie rzeczywistym, analizę częstotliwości błędów oraz automatyczne powiadamianie zespołu o nowych incydentach (np. przez Slack lub e-mail). Sentry działa zarówno jako usługa SaaS, jak i w wersji self-hosted. Przechwytuje automatycznie nieobsłużone wyjątki, ale można także raportować je ręcznie.

Podstawowa konfiguracja (np. w config/initializers/sentry.rb) może wyglądać tak:

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 123

Dzięki integracji z Sentry błędy nie są tylko zapisywane – są aktywnie monitorowane i łatwiej dostępne dla zespołu odpowiedzialnego za utrzymanie aplikacji.

## A10:2021 - Server-Side Request Forgery

#### Opis

Server-Side Request Forgery (SSRF) to podatność polegająca na tym, że serwer aplikacji zostaje zmuszony do wykonania żądania HTTP do wskazanego przez atakującego zasobu – wewnętrznego lub zewnętrznego. Może to prowadzić m.in. do ujawnienia zasobów wewnętrznych (np. paneli administracyjnych, lokalnych API), skanowania infrastruktury, a nawet do wykonania złośliwego kodu.

W Ruby on Rails podatność może wystąpić, gdy aplikacja umożliwia użytkownikowi podanie adresu URL (np. do podglądu obrazka, załadowania treści czy integracji z API), a następnie wykonuje żądanie przy użyciu bibliotek takich jak Net::HTTP czy Faraday, bez weryfikacji docelowego adresu. Brak walidacji może umożliwić ataki SSRF nawet wewnątrz prywatnej sieci serwera.

#### Przykład

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

Aplikacja pozwala użytkownikowi na podanie adresu URL, z którego pobierze fakturę, a nastepnie przetworzy ją i wyświetli kluczowe informacje użytkownikowi. Na poniższym zrzucie ekranu widzimy wszystkie faktury w systemie.

Obraz zawierający tekst, oprogramowanie, Ikona komputerowa, System operacyjny

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 124

##### Poprawne działanie aplikacji

Po kliknięciu przycisku „New Invoice” użytkownik może dodać nową fakturę, wskazując URL, z którego aplikacja pobierze dane. W tym scenariuszu adresy URL są losowymi endpointami zwracającymi JSON – symulują rzeczywiste faktury w formacie PDF.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, numer

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 125

Po przetworzeniu faktury aplikacja prezentuje pobrane dane – w formie surowej oraz po „przetworzeniu” (dla uproszczenia jest to tylko komunikat informacyjny).

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Strona internetowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 126

##### Przykład ataku

Atakujący może wykorzystać tę funkcjonalność do sprawdzenia dostępności wewnętrznych usług. Przykładowo, jeśli aplikacja działa w środowisku Docker Compose lub Kubernetes, to może mieć dostęp do serwisów działających wyłącznie wewnątrz klastra.

W naszym przykładzie mamy usługę internal-app, skonfigurowaną w pliku  
docker-compose.yml, do której nie można się w żaden sposób dostać z zewnątrz klastra:

Obraz zawierający tekst, Czcionka, linia, biały

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 127

Podczas dodawania faktury atakujący podaje jako źródło adres <http://internal-app>:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 128

Po zapisaniu formularza aplikacja wykonuje żądanie do tej usługi i wyświetla jej odpowiedź – oznacza to, że wewnętrzna usługa została odsłonięta na zewnątrz, co stanowi klasyczny przypadek podatności SSRF:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek 129

#### Skutki

Konsekwencje udanego ataku SSRF mogą być bardzo poważne, szczególnie w środowiskach chmurowych i mikrousługowych. Do najczęstszych skutków należą:

* **Ujawnienie wewnętrznych usług** – aplikacja może umożliwić dostęp do zasobów niedostępnych z zewnątrz, takich jak wewnętrzne API, bazy danych czy panele administracyjne.
* **Eksfiltracja metadanych chmurowych** – w środowiskach takich jak AWS możliwe jest pobranie tokenów dostępowych z adresów takich jak http://169.254.169.254, co może prowadzić do przejęcia konta lub eskalacji uprawnień.
* **Skanowanie i mapowanie infrastruktury sieciowej** – atakujący może wykorzystać podatność do odkrywania innych usług działających w sieci wewnętrznej, co ułatwia dalsze ataki lateralne.

#### Zalecenia

##### Walidacja adresów URL pochodzących od użytkownika

Aplikacja powinna weryfikować, czy adresy URL przekazywane przez użytkownika pochodzą z dozwolonej listy domen (whitelist). Należy unikać wykonywania zapytań HTTP do dowolnych, niezweryfikowanych źródeł. Przekazywane adresy powinny być poddane parsowaniu oraz walidacji przed ich użyciem, co pozwala ograniczyć dostęp wyłącznie do zaufanych zasobów zewnętrznych.

##### Blokowanie żądań do lokalnych i prywatnych adresów IP

Aplikacja nie powinna umożliwiać wykonywania zapytań HTTP do lokalnych hostów ani prywatnych adresów IP (np. 127.0.0.1, localhost, 169.254.169.254, 192.168.0.0/16). Przed wykonaniem żądania należy rozwiązać host na adres IP i sprawdzić, czy nie należy on do wewnętrznych zakresów sieciowych. Ochrona ta zabezpiecza m.in. przed dostępem do usług metadanych chmurowych oraz wewnętrznych serwisów infrastruktury.

##### Wydzielenie logiki sieciowej do osobnych usług

W przypadku funkcjonalności polegających na pobieraniu zasobów z internetu, zaleca się wydzielenie tej odpowiedzialności do osobnych mikroserwisów lub workerów, działających w izolowanym środowisku bez dostępu do wewnętrznych zasobów sieciowych aplikacji. Taki podział ogranicza zakres potencjalnego ataku SSRF do wybranego komponentu systemu.

# Zakończenie

W niniejszej pracy magisterskiej przedstawiono praktyczne ujęcie zagrożeń bezpieczeństwa aplikacji webowych na przykładzie frameworka Ruby on Rails, zgodnie z klasyfikacją OWASP Top 10. Każda z omawianych podatności została zaprezentowana nie tylko w formie teoretycznego opisu, ale również zilustrowana działającym przykładem, co pozwoliło lepiej zrozumieć mechanizmy działania zagrożeń i ich potencjalne konsekwencje.

Duży nacisk położono na praktykę – omówiono nie tylko objawy i skutki luk, ale również sposoby ich identyfikacji, typowe błędy konfiguracyjne oraz zalecenia dotyczące bezpiecznego programowania w środowisku Rails. Dzięki temu praca może służyć zarówno jako przegląd zagrożeń, jak i praktyczny poradnik dla programistów i inżynierów bezpieczeństwa.

Podczas analizy zwrócono również uwagę na aspekty infrastrukturalne i środowiskowe, takie jak różnice w konfiguracji aplikacji zależnie od sposobu wdrożenia (np. VPS vs Kubernetes), co czyni podejście do bezpieczeństwa bardziej całościowym. Omawiane zagrożenia i środki zaradcze mogą być z powodzeniem adaptowane także w innych frameworkach czy językach programowania.

Mimo że ochrona przed wszystkimi możliwymi wektorami ataku jest trudna, stosowanie dobrych praktyk, świadomego podejścia do konfiguracji oraz regularne aktualizacje oprogramowania znacząco zwiększają odporność aplikacji. Mam nadzieję, że przedstawiony materiał okaże się przydatny i inspirujący zarówno dla osób rozpoczynających przygodę z bezpieczeństwem, jak i dla tych bardziej doświadczonych, poszukujących konkretnych rozwiązań w codziennej pracy.

# Bibliografia

# Wykaz rysunków

[Rysunek 1 16](#_Toc195952310)

[Rysunek 2 16](#_Toc195952311)

[Rysunek 3 17](#_Toc195952312)

[Rysunek 4 17](#_Toc195952313)

[Rysunek 5 18](#_Toc195952314)

[Rysunek 6 19](#_Toc195952315)

[Rysunek 7 19](#_Toc195952316)

[Rysunek 8 19](#_Toc195952317)

[Rysunek 9 19](#_Toc195952318)

[Rysunek 10 20](#_Toc195952319)

[Rysunek 11 21](#_Toc195952320)

[Rysunek 12 21](#_Toc195952321)

[Rysunek 13 22](#_Toc195952322)

[Rysunek 14 22](#_Toc195952323)

[Rysunek 15 23](#_Toc195952324)

[Rysunek 16 24](#_Toc195952325)

[Rysunek 17 25](#_Toc195952326)

[Rysunek 18 25](#_Toc195952327)

[Rysunek 19 26](#_Toc195952328)

[Rysunek 20 28](#_Toc195952329)

[Rysunek 21 29](#_Toc195952330)

[Rysunek 22 30](#_Toc195952331)

[Rysunek 23 31](#_Toc195952332)

[Rysunek 24 32](#_Toc195952333)

[Rysunek 25 33](#_Toc195952334)

[Rysunek 26 34](#_Toc195952335)

[Rysunek 27 35](#_Toc195952336)

[Rysunek 28 36](#_Toc195952337)

[Rysunek 29 36](#_Toc195952338)

[Rysunek 30 38](#_Toc195952339)

[Rysunek 31 38](#_Toc195952340)

[Rysunek 32 38](#_Toc195952341)

[Rysunek 33 38](#_Toc195952342)

[Rysunek 34 39](#_Toc195952343)

[Rysunek 35 39](#_Toc195952344)

[Rysunek 36 39](#_Toc195952345)

[Rysunek 37 40](#_Toc195952346)

[Rysunek 38 40](#_Toc195952347)

[Rysunek 39 40](#_Toc195952348)

[Rysunek 40 41](#_Toc195952349)

[Rysunek 41 44](#_Toc195952350)

[Rysunek 42 44](#_Toc195952351)

[Rysunek 43 44](#_Toc195952352)

[Rysunek 44 45](#_Toc195952353)

[Rysunek 45 45](#_Toc195952354)

[Rysunek 46 46](#_Toc195952355)

[Rysunek 47 46](#_Toc195952356)

[Rysunek 48 46](#_Toc195952357)

[Rysunek 49 49](#_Toc195952358)

[Rysunek 50 49](#_Toc195952359)

[Rysunek 51 50](#_Toc195952360)

[Rysunek 52 50](#_Toc195952361)

[Rysunek 53 50](#_Toc195952362)

[Rysunek 54 51](#_Toc195952363)

[Rysunek 55 51](#_Toc195952364)

[Rysunek 56 51](#_Toc195952365)

[Rysunek 57 52](#_Toc195952366)

[Rysunek 58 52](#_Toc195952367)

[Rysunek 59 53](#_Toc195952368)

[Rysunek 60 56](#_Toc195952369)

[Rysunek 61 56](#_Toc195952370)

[Rysunek 62 56](#_Toc195952371)

[Rysunek 63 57](#_Toc195952372)

[Rysunek 64 57](#_Toc195952373)

[Rysunek 65 57](#_Toc195952374)

[Rysunek 66 57](#_Toc195952375)

[Rysunek 67 58](#_Toc195952376)

[Rysunek 68 58](#_Toc195952377)

[Rysunek 69 59](#_Toc195952378)

[Rysunek 70 59](#_Toc195952379)

[Rysunek 71 61](#_Toc195952380)

[Rysunek 72 61](#_Toc195952381)

[Rysunek 73 62](#_Toc195952382)

[Rysunek 74 63](#_Toc195952383)

[Rysunek 75 63](#_Toc195952384)

[Rysunek 76 64](#_Toc195952385)

[Rysunek 77 64](#_Toc195952386)

[Rysunek 78 64](#_Toc195952387)

[Rysunek 79 65](#_Toc195952388)

[Rysunek 80 65](#_Toc195952389)

[Rysunek 81 65](#_Toc195952390)

[Rysunek 82 66](#_Toc195952391)

[Rysunek 83 68](#_Toc195952392)

[Rysunek 84 68](#_Toc195952393)

[Rysunek 85 69](#_Toc195952394)

[Rysunek 86 69](#_Toc195952395)

[Rysunek 87 70](#_Toc195952396)

[Rysunek 88 71](#_Toc195952397)

[Rysunek 89 71](#_Toc195952398)

[Rysunek 90 72](#_Toc195952399)

[Rysunek 91 72](#_Toc195952400)

[Rysunek 92 73](#_Toc195952401)

[Rysunek 93 73](#_Toc195952402)

[Rysunek 94 74](#_Toc195952403)

[Rysunek 95 76](#_Toc195952404)

[Rysunek 96 76](#_Toc195952405)

[Rysunek 97 77](#_Toc195952406)

[Rysunek 98 79](#_Toc195952407)

[Rysunek 99 79](#_Toc195952408)

[Rysunek 100 81](#_Toc195952409)

[Rysunek 101 82](#_Toc195952410)

[Rysunek 102 82](#_Toc195952411)

[Rysunek 103 84](#_Toc195952412)

[Rysunek 104 84](#_Toc195952413)

[Rysunek 105 86](#_Toc195952414)

[Rysunek 106 87](#_Toc195952415)

[Rysunek 107 87](#_Toc195952416)

[Rysunek 108 88](#_Toc195952417)

[Rysunek 109 91](#_Toc195952418)

[Rysunek 110 91](#_Toc195952419)

[Rysunek 111 91](#_Toc195952420)

[Rysunek 112 92](#_Toc195952421)

[Rysunek 113 92](#_Toc195952422)

[Rysunek 114 93](#_Toc195952423)

[Rysunek 115 93](#_Toc195952424)

[Rysunek 116 94](#_Toc195952425)

[Rysunek 117 94](#_Toc195952426)

[Rysunek 118 94](#_Toc195952427)

[Rysunek 119 96](#_Toc195952428)

[Rysunek 120 96](#_Toc195952429)

[Rysunek 121 97](#_Toc195952430)

[Rysunek 122 97](#_Toc195952431)

[Rysunek 123 97](#_Toc195952432)

[Rysunek 124 98](#_Toc195952433)

[Rysunek 125 99](#_Toc195952434)

[Rysunek 126 99](#_Toc195952435)

[Rysunek 127 100](#_Toc195952436)

[Rysunek 128 100](#_Toc195952437)

[Rysunek 129 101](#_Toc195952438)

**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**

-

[Zdjęcie z oświadczeniem]