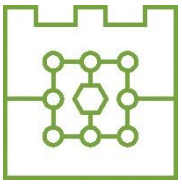
Politechnika Krakowska

im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji



Michał Mazur

Numer albumu: 148907

Aspekty bezpieczeństwa testowego API w Spring

Security aspects of the test API in Spring

Praca magisterska

na kierunku Informatyka

specjalność Cyberbezpieczeństwo

Praca wykonana pod kierunkiem:
dr Radosław Kycia

Kraków, 2024

Spis treści

Streszczenie

Niniejsza praca magisterska skupia się na badaniu aspektów bezpieczeństwa aplikacji opartej o interfejs API zrealizowanej w technologii Spring Boot. Celem pracy było zrealizowanie testowego interfejsu API, który został poddany testom bezpieczeństwa wykrywającym podatności na ataki cybernetyczne. Dodatkowo na podstawie wykrytych luk zostały wprowadzone udoskonalenia niwelujące wykryte podatności.

Badanie zostało podzielone na dwa etapy – aktywne skanowanie i testy manualne potwierdzające wykryte podatności za pomocą skanu. Dodatkowo zostały przeprowadzone testy sprawdzające mechanizm uwierzytelniania użytkowników poprzez próbę dekodowania tokenu JWT oraz sprawdzenie odporności na atak metodą siły. Z wykorzystaniem skanowania oraz testów manualnych zostało wykryte pięć podatności, które nie są wynikami fałszywie pozytywnymi lecz faktycznie występują w testowym interfejsie.

Na podstawie wykrytych podatności zostały wprowadzone poprawki w testowym interfejsie, których celem jest zniwelowanie znalezionych podatności podczas badania. Z uwagi na zaimplementowane mechanizmy niwelujące wykryte podatności ponowne aktywne skanowanie aplikacji nie wykryło żadnych nowych oraz wcześniej zdiagnozowanych luk w interfejsie, dzięki czemu cel pracy został osiągnięty.

Abstract

This thesis focuses on the study of the security aspects of an API-based application realized with Spring Boot technology. The purpose of the thesis was to realize a test API, which was subjected to security testing to detect vulnerabilities to cyber-attacks. In addition, based on the detected vulnerabilities, improvements were made to mitigate the detected vulnerabilities.

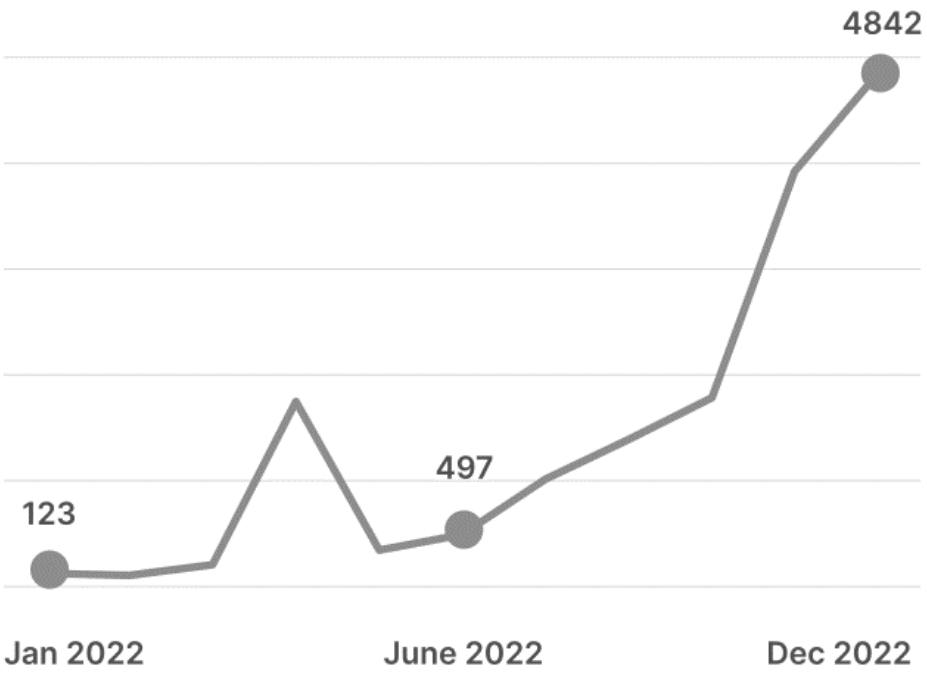
The survey was divided into two stages - active scanning and manual testing to confirm the detected vulnerabilities through the scan. In addition, tests were carried out to verify the authentication mechanism of users by attempting to decode the JWT token, as well as to check the resistance to a brute force attack. Using scanning and manual testing, five vulnerabilities were detected, which are not false positives but actually present in the test interface.

Based on the detected vulnerabilities, improvements were made to the test interface to mitigate the vulnerabilities found during the test. Due to the implemented mechanisms for levelling the detected vulnerabilities, a new active scan of the application did not detect any new and previously diagnosed vulnerabilities in the interface, so the goal of the work was achieved.

1. Wstęp

W obecnych czasach interfejs API (ang. Application Programming Interface) jest coraz częściej wykorzystywany między innymi przez aplikacje webowe czy różnego rodzaju systemy informatyczne oraz zazwyczaj opiera się na architekturze mikroserwisów, w którym całe API jest podzielone na mniejsze części kodu realizujące inne zadania. Szacowany przez analityków ruch w sieci generowany przez API przekracza 80% całego ruchu w Internecie [1], przy czym większość z takich aplikacji nie jest dobrze przetestowana pod kątem bezpieczeństwa i posiada potencjalne luki umożliwiające na przykład wykradanie wrażliwych danych.

Wraz ze wzrostem popularności interfejsów API, ich znaczenie jest większe ze względu na dużą ilość generowanego przez nie ruchu w sieci. Co z kolei przekłada się na zwiększenie potencjalnych ataków cybernetycznych. Raport firmy Salt Labs z działu Salt Security [8] pokazuje, że interfejsy API ulegają różnym atakom cybernetycznym. Na podstawie danych ich kontrahentów z roku 2022 pokazany jest znaczny wzrost incydentów cybernetycznych względem API. W samym grudniu 2022 roku liczba odnotowanych incydentów wynosiła 4842, co oznacza wzrost na poziomie około czterystu procent do ubiegłych miesięcy i możemy to zobaczyć na Rys 1.



Rys 1 Dane klientów Salt: Atakujący interfejsy API klientów w roku 2022 [8]

Jak wynika z ich raportu około dziewięćdziesiąt cztery procent ich kontrahentów na przestrzeni roku 2022 miało problemy związane z bezpieczeństwem interfejsów API, które są dostępne dla użytkowników końcowych. Jednym z największych problemów związanych z bezpieczeństwem, bo aż czterdzieści jeden procent to luki w zabezpieczeniach, następnie były problemy z mechanizmem uwierzytelniania około czterdziestu procent, a pozostałe punkty procentowe odnosiły się bezpośrednio do incydentów bezpieczeństwa związanych z danymi wrażliwymi czy problemami, które generowały ogromne koszty.

1.1. Motywacja

Pierwszym ważnym aspektem jest fakt, że w dzisiejszym dynamicznym rozwoju oprogramowania, korzystanie z interfejsów API stało się bardzo popularne podczas tworzenia aplikacji internetowych, ponieważ jak wcześniej wspominałem generuje ona ponad osiemdziesiąt procent całego ruchu w sieci. Natomiast wraz ze zwiększającą się złożonością takich systemów zwiększa się również stopień zabezpieczenia takich interfejsów przed potencjalnymi zagrożeniami. Ze względu na to chciałbym poszerzyć swoją wiedzę w zakresie zabezpieczania aplikacji internetowych od strony serwerowej oraz nabyć wiedzę odnoszącą się do testowania aplikacji pod kątem bezpieczeństwa.

Drugim bezpośrednim personalnym powodem jest to, że wiążę przyszłość z technologią Spring, dlatego chciałbym poszerzyć swoją wiedzę na temat sposobów realizowania zabezpieczeń w tej technologii oraz jakie są możliwe drogi realizacji takich zabezpieczeń. Dodatkowo jest to temat, który dobrze konfrontuje się z aktualnymi wyzwaniami dotyczącymi cyberbezpieczeństwa, w którym ataki na interfejsy API są coraz częstsze i złożone.

Współczesne aplikację oparte na np.: architekturze mikroserwisów, ale i nie tylko, często wykorzystują interfejsy API, które mogą generować potencjalne ryzyka bezpieczeństwa. Na podstawie tego chciałbym się nauczyć jak zidentyfikować, zrozumieć oraz rozwiązywać problemy, które związane są z bezpieczeństwem API, aby skutecznie chronić aplikację przed potencjalnymi atakami.

1.2. Cel pracy

Z uwagi na wcześniej wymienione kwestie, bezpieczeństwo stanowi kluczowy element serwerów opartych o interfejs API. Natomiast nie jesteśmy w stanie określić czy dana aplikacją bądź system informatyczny są całkowicie odporne na ataki cybernetyczne, ponieważ pojęcie „Bezpieczeństwa” jest względne i cały czas znajdowane są nowe luki w zabezpieczeniach. W związku z tym w niniejszej pracy postaram się zaprezentować sposób testowania aplikacji

opartej na interfejsie API w celu wykrycia potencjalnych zagrożeń cybernetycznych, ponadto omówię elementy jakie należy uwzględnić, aby w aplikacji osiągnąć wysoki stan zabezpieczeń.

Przedmiotem badań jest identyfikacja, zrozumienie oraz przede wszystkim analiza różnych aspektów bezpieczeństwa testowego API w Spring. Natomiast głównym celem jest znalezienie potencjalnych luk oraz zrealizowanie skutecznych praktyk strategii obrony przed atakami i potencjalnymi zagrożeniami oraz wdrożenie ich na podstawie wyników z praktycznej części pracy.

1.3. Metodyka

Główną metodą badawczą jest przeprowadzenie eksperymentów praktycznych np.: testów penetracyjnych lub aktywnych skanów, w celu zidentyfikowania potencjalnych luk w aplikacji testowej. Badania dotyczące aspektów bezpieczeństwa testowego API w Spring nie mają na celu, tylko wyłącznie identyfikacji potencjalnych podatności, ale również na wprowadzeniu zabezpieczeń przeciwko nim. Dodatkowo na podstawie badań zostaną opracowane skuteczne środki zaradcze przeciwko wykrytym podatnościom.

1.4. Zakres pracy

Przedmiotem niniejszej pracy magisterskiej jest zrealizowanie testowego serwera opartego na interfejsie API, który zostanie poddany testom bezpieczeństwa oraz aktywnemu skanowaniu za pomocą specjalistycznych narzędzi w celu znalezienia potencjalnych luk, a następnie zneutralizowanie wykrytych podatności. Omówione zostaną różne aspekty bezpieczeństwa wraz z praktycznymi testami odnoszącymi się do zabezpieczeń aplikacji opartych na API oraz zostaną zaprezentowane rekomendację i środki zaradcze w celu tworzenia bezpiecznych aplikacji internetowych.

1.5. Zarys pracy

Praca ta skupiać się będzie bezpośrednio na kluczowych elementach bezpieczeństwa, które wymagają testowania oraz na elementach jakie trzeba wdrożyć do serwera opartego na interfejsie API. Nie będzie natomiast omawiać procesu inżynierii oprogramowania ani prezentować sposobów tworzenia API od podstaw. Pierwszy rozdział zostanie poświęcony na omówienie tematyki niniejszej pracy magisterskiej. Obejmować to będzie motywację do wyboru tematyki, metodykę badań oraz cel i zakres pracy. Kolejne rozdziały od drugiego do czwartego będą skupiały się na teoretycznych aspektach niniejszej pracy, przedstawiając między innymi kluczowe terminy technologiczne, potencjalne zagrożenia dla aplikacji API,

przykładowe incydenty związane z brakami w zabezpieczeniach oraz etapy przeprowadzania testów wraz z najistotniejszymi narzędziami służącymi do testowania bezpieczeństwa. Kolejnym etapem pracy będzie część praktyczna rozpoczynająca się od rozdziału piątego, który będzie prezentował przygotowanie wirtualnego środowiska przeznaczonego do realizacji badań. W kolejnych rozdziałach zaprezentowane będą testy bezpieczeństwa składające się z aktywnego skanowania oraz testów manualnych, dodatkowo wykryte podatności zostaną sklasyfikowane oraz sporządzone będą środki zaradcze, na podstawie których zaimplementowane zostaną mechanizmy bezpieczeństwa likwidujące wykryte podatności. Ostatni rozdział będzie poświęcony na podsumowanie oraz refleksję na temat przeprowadzonego procesu badawczego.

Część Teoretyczna

2. Podstawy technologiczne

W drugim rozdziale przedstawione zostaną wszystkie niezbędne podstawy technologiczne lub definicje, w celu bezproblemowego zrozumienia niniejszej pracy magisterskiej.

2.1. API - Application Programming Interface

API (ang. Application Programming Interface) najprościej mówiąc jest to zbiór ustalonych zasad i protokołów, które umożliwiają komunikowanie się pomiędzy różnymi elementami oprogramowania bądź systemu, ale i nie tylko, ponieważ do API mogą mieć dostęp również zewnętrzni klienci. Za pomocą tego rozwiązana serwer oparty o interfejs API udostępnia dane lub informację na żądania klientów. Interfejs ten posiada trzy główne rodzaje REST, GraphQL oraz SOAP, które możemy wykorzystać przy jego tworzeniu.

Pierwszym z nich jest *Representational State Transfer* (REST) [1]. To standard oraz jeden z najczęściej wybieranych styli architektonicznych tworzenia API. Określa on komunikację pomiędzy aplikacjami wykorzystując protokół HTTP. Powiązanym pojęciem opisującym model dojrzałości interfejsów API opartych na standardzie REST jest model dojrzałości Richardsona. Przedstawia etapy dojrzałości interfejsu za pomocą czterech poziomów. Im dany interfejs API posiada wyższy poziom tym bardziej jest on zgodny ze standardem. Poziom zerowy odnosi się do interfejsu, który wykorzystuję tylko jeden adres oraz nie wykorzystuję metod HTTP. Poziom pierwszy – zasoby (ang. Resources) posiadają interfejsy, które udostępniają zasoby z wykorzystaniem konkretnych adresów, przy czym w dalszym ciągu nie wdrożone zostały różne metody HTTP. Poziom drugi - czasowniki HTTP (ang. HTTP Verbs) skierowany jest do API, które wykorzystują różne metody HTTP oraz udostępniają zasoby z wykorzystaniem stałych adresów, w których zawarte są czasowniki. Ostatni poziom trzeci – kontrola hipermediów (ang. Hipermedia Controls) przypisywany jest do interfejsów, w których zaimplementowany został hipertekst jako silnik stanu aplikacji (ang. Hypertext As The Engine of Application State), który dołącza do odpowiedzi potencjalne akcje jakie mogą zostać zrealizowane na dostarczonym zasobie. REST narzuca pewne wzorce i dobre praktyki określające format wymiany danych. Polega on na wykorzystywaniu adresów URL, w celu odpowiadania na żądania protokołu HTTP. REST API zazwyczaj obsługuję swoich klientów z wykorzystaniem formatu JSON lub XML. Sam w sobie nie zapewnia możliwości szyfrowania danych, niemniej jednak możemy to zrealizować za pomocą wdrożenia na

przykład certyfikatu SSL (ang. Security Sockets Layer) do protokołu HTTP, dzięki czemu otrzymamy zabezpieczenie przesyłanych danych między klientem a serwerem.

Graph Query Language (GraphQL) [1] to standard, który polega na definiowaniu przez klienta struktur żądanych danych. Standard ten skupia się na zapytaniach, które podobne są do zapytań SQL oraz określają strukturę danych za pomocą grafu. Wykorzystuję on także protokół HTTP z tylko jednym adresem URL i metodą POST. W porównaniu do REST, dzięki umożliwieniu elastycznych „żądań" jest on dynamiczniejszy i bardziej intuicyjny. Posiada jeden kluczowy aspekt, który sprawia, że jest lepszy od standardu REST, a mianowicie GraphQL umożliwia pobieranie konkretnych danych wykorzystując strukturę grafu, w odróżnieniu do REST, który zwraca predefiniowane dane oraz często należy wykonać kilka zapytań, aby dostać ściśle określone informację.

Simple Object Access Protocol (SOAP) [1] jest protokołem komunikacyjnym opartym na formacie XML, który był jednym z pierwszych sposobów komunikacyjnych między usługami. Najczęściej wykorzystuję protokół HTTP do wymiany informacji, natomiast umożliwia również wykorzystanie innych protokołów komunikacyjnych takich jak SMTP, TCP lub UDP. W odróżnieniu od dwóch pierwszych, SOAP jest protokołem, który umożliwia szyfrowanie danych, natomiast nie posiada on wysokiej elastyczności.

2.2. Spring – wprowadzenie

Spring jest popularnym frameworkiem z otwartym kodem źródłowym, którego pionierem jest Rod Johnson. Umożliwia on tworzenie aplikacji webowych w języku java, czyli aplikacje klasy korporacyjnej. Powstał z myślą o uproszczeniu tworzenia aplikacji ze względu na złożoność oraz nieintuicyjność technologii komponentów Enterpise Java (ang. Enterpise JavaBeans). Spring wprowadza wiele udogodnień dla programistów dzięki czemu zyskał on dużą popularność oraz posiada szereg wspomagających go komponentów bądź bibliotek. Jego głównymi zaletami funkcjonalnymi są: *wstrzykiwanie zależności* (ang. Dependency Injection) oraz *programowanie zorientowane na aspekty* (ang. Aspect-Oriented Programming). Ponadto umożliwia nie tylko tworzenie serwerów opartych na języku java, ale również tworząc zwykłe aplikację we wcześniej wspomnianym języku możemy korzystać z funkcjonalności springa w postaci między innymi testowania czy powiązań między obiektami [3].

Niestety sam Spring wymaga wielu konfiguracji oraz dużej ilości kodu szablonowego, który za każdym razem programista musiał zrealizować. Na tej podstawie narodziła się kolejna idea z rodziny Spring, czyli *Spring Boot*. Jest to projekt, który umożliwia budowanie aplikacji

na podstawie springa przy wykorzystaniu automatycznych mechanizmów konfiguracji oraz małej ilości szablonowego kodu. Największe korzyści jakie płyną ze Spring Boota to łatwość uruchamiania, automatyczna konfiguracja oraz szybkość tworzenia aplikacji [4].

2.3. Bezpieczeństwo w Spring Boot

Aspekt bezpieczeństwa w Spring Boot realizowany jest między innymi z wykorzystaniem dedykowanego frameworka Spring Security bazującego bezpośrednio na Spring, dzięki czemu możliwe jest wykorzystywanie głównych aspektów frameworka jakim jest np.: wstrzykiwania zależności [3].

Spring Security jest kluczowym elementem zabezpieczeń oferujący zróżnicowane podejścia do rozwiązań odnoszących się bezpośrednio do kwestii związanych między innymi z autoryzacją oraz uwierzytelnianiem. Oferuję on wiele funkcji oraz mechanizmów, między innymi do najpopularniejszych z nich należy:

Uwierzytelnianie (ang. Authentication) użytkowników poprzez stosowanie różnych metod, takich jak formularze logowania, uwierzytelnianie za pomocą OAuth lub za pomocą tokenów na przykład JSON Web Token.

Drugim mechanizmem jest autoryzacja (ang. Authorization), zapewnia ona metody umożliwiające kontrolę dostępu do zasobów, aby określić jakie punkty końcowe mogą być dostępne dla konkretnej grupy użytkowników lub danego użytkownika. Jak większość rzeczy w Spring jest to możliwe poprzez adnotacje, które należy umieścić nad konkretną metodą lub w klasie konfiguracyjnej.

Posiada również filtry serwletów, które są komponentami umożliwiającymi przechwytywanie i przetwarzanie żądań oraz odpowiedzi zanim zostaną przekazane lub odesłane z serwletu – klasy odpowiedzialnej za rozszerzenie możliwości obsługi żądań klienta [3]. Umożliwia to obsługę różnych aspektów zabezpieczeń, takich jak uwierzytelnianie, zarządzanie sesją czy zapewnienie mechanizmu obrony przed atakami typu Cross-Site Request Forgery, który będzie przedstawiony w późniejszym rozdziale 3 [3].

Dodatkowo podczas implementowania interfejsu API opartego o technologię Spring Boot często wykorzystywana jest warstwa danych realizowana za pomocą framework *Hibernate* [4], który implementuje moduł *JPA* (ang. Java Persistence API). Hibernate dostarcza pewnych mechanizmów uniemożliwiających na przykład ataki typu wstrzykiwania złośliwego kodu SQL. Framework zarządza bazą danych między innymi przy pomocy metod zapytania

(ang. Query Method) lub języka zapytań Hibernate (ang. Hibernate Query Language). Oba sposoby odwoływania się do bazy danych korzystają z parametryzowanych zapytań, dzięki czemu w pewnym stopniu niwelują potencjalną podatność na wstrzykiwanie złośliwego kodu SQL lub skryptów między witrynami.

2.4. Skanowanie podatności

Skanowanie w poszukiwaniu podatności jest kluczowym elementem testowania bezpieczeństwa aplikacji i systemów. Polega na zidentyfikowaniu ewentualnych podatności oraz luk w zabezpieczeniach za pomocą specjalistycznego automatycznego narzędzia realizującego przeszukiwanie na przykład punktów końcowych pod kątem podatności, które zawarte są w bazie danych wykorzystywanego narzędzia. Jest to proces, który nie gwarantuje stu procentowej pewności w znalezieniu wszystkich podatności ze względu na to, że wykorzystuję skaner, który bazuje na zbiorze znanych podatności i luk. Wyniki skanowania są uzależnione od rodzaju skanera jaki jest wykorzystany w danym oprogramowaniu do wykrywania zagrożeń. Warto zauważyć, że wiele z tych narzędzi generuje fałszywie pozytywne lub fałszywie negatywne wyniki, co stanowi istotne ryzyko, ponieważ mogą one nie wykryć pewnych podatności lub błędnie je zidentyfikować.

W przypadku fałszywie pozytywnego wyniku, odkrycie danej podatności przez skaner nie oznacza koniecznie, że będzie ona wykorzystana przez potencjalnego cyberprzestępcę, ponieważ w istocie może jej wcale nie być. Możemy dodatkowo manualnie zweryfikować tę podatność, co jest istotne. Z kolei fałszywie negatywny wynik oznacza, że test, który przeszedł pozytywnie skanowanie, nie oznacza, że danej podatności nie ma w badanym podmiocie. Jest to skrajnie niebezpieczne z uwagi nieświadomości, że dana podatność może jednak występować. Oba te scenariusze mogą stanowić problem w ocenie ryzyka bezpieczeństwa dla badanych systemów i aplikacji [5].

2.5. Testy penetracyjne – definicja i cele

Testy penetracyjne [5] w dalszej części pracy nazywane testami manualnymi są nieodłącznym elementem testowania aplikacji pod kątem cyberbezpieczeństwa. Polegają one na wykorzystaniu znalezionych wcześniej podatności oraz luk w bezpieczny sposób, aby zrealizować monitorowany atak na aplikację lub system bez wyrządzania potencjalnego ryzyka stabilności działania podmiotu na przykład poprzez wykorzystanie zwirtualizowanego środowiska testowego. Realizowanie takich testów daje potencjalnie mniejszy współczynnik występowania fałszywie pozytywnych wyników, związane jest to, że do testów

wykorzystywane są wszystkie wcześniej znalezione podatności. Jedną z kluczowych wad testów penetracyjnych jest to, że do ich realizacji wykorzystane są znane podatności lub fragmenty kodu wykorzystujące daną podatność (ang. Exploit), przez co nie jest możliwe określenie, że testy dają faktyczny obraz zabezpieczeń.

Celem testów penetracyjnych [5] jest zidentyfikowanie i zabezpieczenie potencjalnych luk w badanym podmiocie. Pomagają one w identyfikacji podatności takich jak niewystarczająca autoryzacja, uwierzytelnianie, braki w kontroli dostępu czy też możliwości ataków wstrzykiwania złośliwego kodu SQL (ang. SQL Injection), skrypty między witrynami (ang. Cross-Site Scripting) czy sfałszowane żądania między witrynami (ang. Cross-Site Request Forgery). Wszystkie te podatności zostaną szczegółowo przedstawione w rozdziale 3.

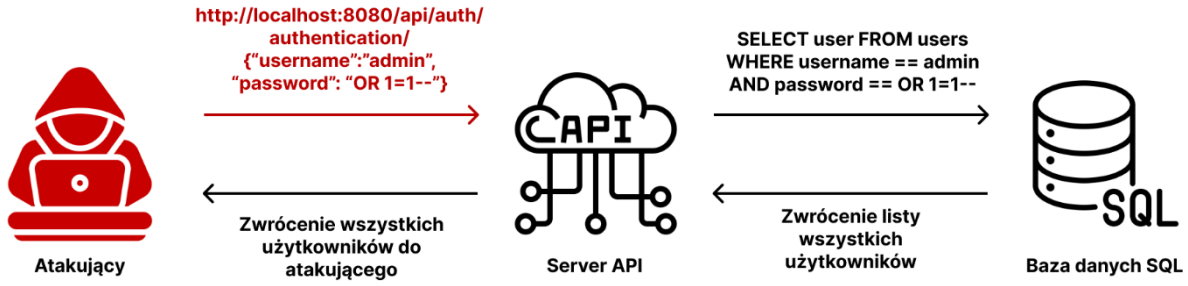
3. Analiza zagrożeń i podatności

Rozdział ten poświęcony jest na przedstawienie najpowszechniejszych podatności jakie mogą występować w aplikacjach internetowych. Dodatkowo omówione zostaną przykładowe incydenty jakie miały miejsce na przestrzeni lat, w związku z na przykład brakami w bezpieczeństwie.

3.1. Typowe zagrożenia dla aplikacji API

Aplikacje tworzone z wykorzystaniem interfejsów API mogą posiadać różne potencjalne zagrożenia, jeżeli nie są dobrze zabezpieczone i przetestowane. Do najpopularniejszych podatności jakie mogą występować w aplikacjach z interfejsem API ale i nie tylko należą między innymi:

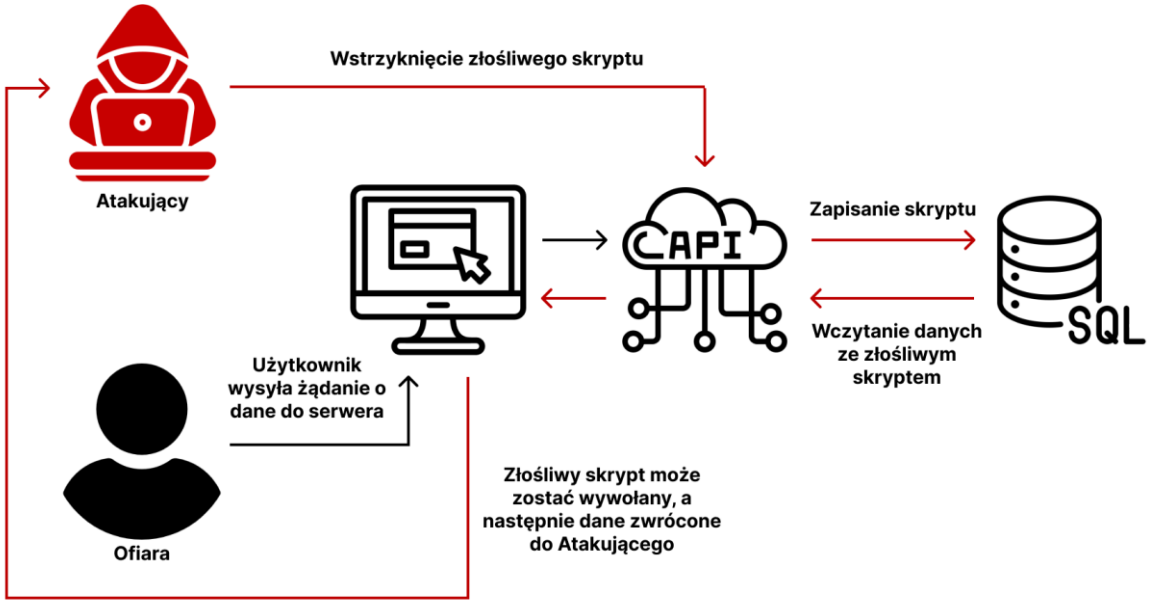
Wstrzykiwanie złośliwego kodu SQL do zapytań (ang. SQL Injection) [1]. Jest to jeden z najpopularniejszych ataków, które są przeprowadzana nie tylko na aplikacje serwerowe oparte na interfejsie API, ale również na inne aplikacje wykorzystujące bazę danych np.: witryny internetowe. Polega na manipulowaniu zapytania poprzez umieszczenie w nim dodatkowych danych. W przypadku aplikacji, które nie posiadają mechanizmu walidacji danych, tego typu działanie może prowadzić do wykonania nieautoryzowanych zmian. Za pomocą ataku tego typu, atakujący może uzyskać dostęp do aplikacji bez autoryzacji, modyfikować oraz usuwać dane z bazy danych lub wykonać kod w bazie danych, który spowoduje nieodwracalne szkody w systemie bazodanowym. Na Rys 2 został przedstawiony schemat ataku opartego na wstrzykiwaniu kodu SQL.



Rys 2 Przykładowy schemat ataku wstrzykiwania złośliwego kod SQL – opracowanie własne

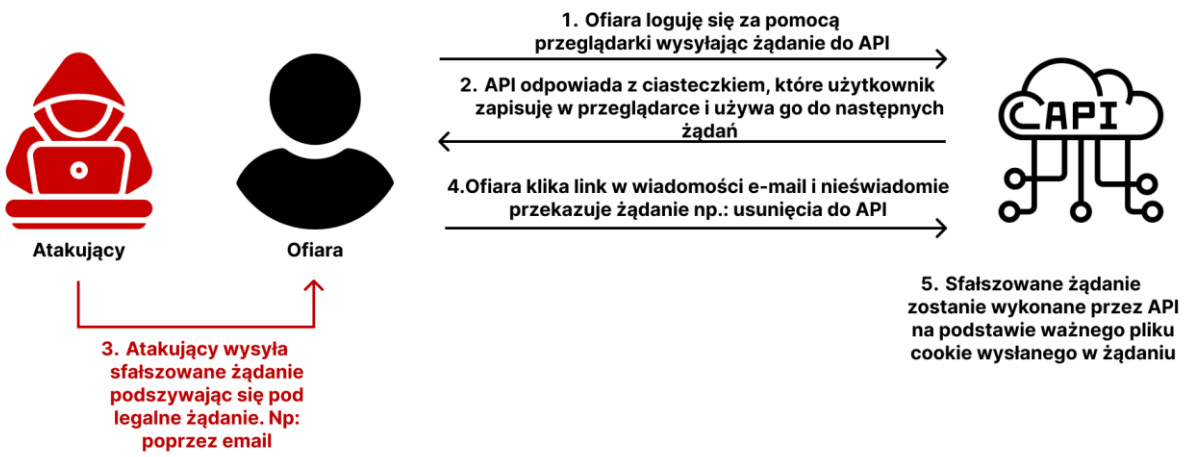
Drugą najpopularniejszą podatnością są *skrypty między witrynami* (ang. Cross-Site Scripting - XSS). Ten rodzaj ataku polega na „wstrzyknięciu" skryptów zazwyczaj napisanych w języku JavaScript, do kodu źródłowy strony internetowej jednocześnie wymuszając na użytkowniku wywołanie go poprzez kliknięcie np.: niebezpiecznego odnośnika. Celem ataku XSS jest wykorzystanie luk w zabezpieczeniach, aby uzyskać dostęp do sesji, danych

użytkownika czy nawet przejąć kontrolę nad kontem. Jednym z rodzajów XSS, który wykorzystuję serwer jest *zapisany atak skryptów między witrynami* (ang. Stored Cross-Site Scripting), polegający na zapisaniu złośliwego kodu po stronie serwerowej np.: w bazie danych, a następnie uruchomienie skryptu, kiedy użytkownik odwiedza stronę internetową [1]. Na Rys 3 zaprezentowano przykładowy schemat procesu ataku typu Cross-Site Scripting.



Rys 3 Przykładowy schemat ataku skryptów między witrynami – opracowanie własne

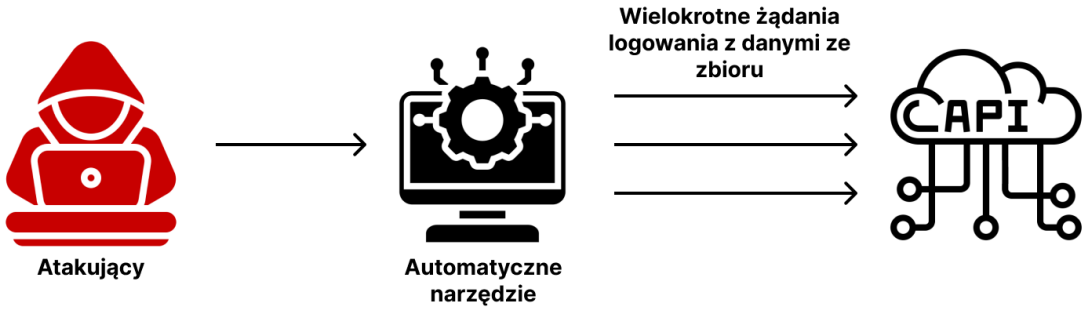
Następną podobną podatnością do poprzedniej jest *sfałszowane żądania między witrynami* (ang. Cross-Site Request Forgery). Atak ten realizowany jest na użytkownikach aplikacji internetowej, który polega na wykonaniu niepożądanych działań w ich imieniu, bez ich zgody oraz wiedzy. Realizowany jest poprzez fałszowanie przez atakującego zapytania i podesłanie go osobie nieświadomie go realizującej. W tego typu ataku głównie chodzi o uruchomienie pewnego odnośnika, który daje atakującemu możliwości jakich wcześniej nie mógł zrealizować np.: pozyskanie danych administratora systemu, aplikacji czy nawet zrealizować przelew bankowy [1]. Rys 4 prezentuje przykładowy przebieg ataku sfałszowanego żądania między witrynami.



Rys 4 Przykładowy schemat ataku sfałszowania żądania między witrynami – opracowanie własne

Czwartą podatnością jest *nadmierna ekspozycja danych* [1]. Zazwyczaj polega to na dołączaniu do odpowiedzi dodatkowych informacji, które są poza zakresem żądania. Wiele interfejsów jest projektowanych z założeniem, że to klient ma możliwość filtrowania otrzymywanych danych i wybierania tych, które go interesują. Ta podatność może również występować w komunikatach o błędach zwracanych przez serwer, na przykład haker może dowiedzieć się, że dany email jest już zarejestrowany, ze względu na opis błędu zawierający informację o istnieniu takiego adresu w systemie.

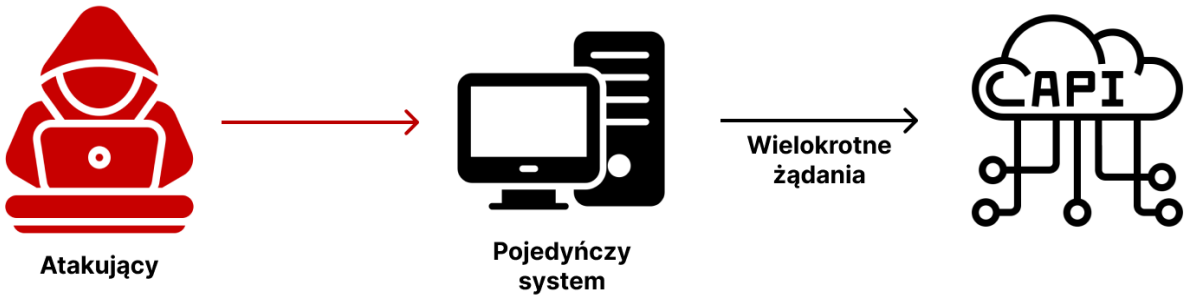
Kolejnym typem podatności jest *metoda siłowa* (ang. Bruteforce) [3]. Jest to jeden z najpopularniejszych ataków tego typu. Jego celem jest uzyskanie dostępu do konta na podstawie zbioru haseł i nazw użytkowników, które mogą pochodzić z wycieku na przykład innego systemu. Polega na wysyłaniu wielu żądań pod konkretny punkt końcowy zawierające dane ze zbioru w odpowiednim formacie JSON lub XML. W celu zwiększenia efektywności tego ataku, atakujący często wykorzystują również poprzednią podatność nadmiernej ekspozycji danych, aby zweryfikować czy dana nazwa użytkownika widnieje w systemie, następnie próbować zalogować się na jego podstawie z hasłami ze zbioru. Za pomocą Rys 5 przedstawiono przykładowy schemat ataku metodą siłową.



Rys 5 Przykładowy schemat ataku metodą siłową – opracowanie własne

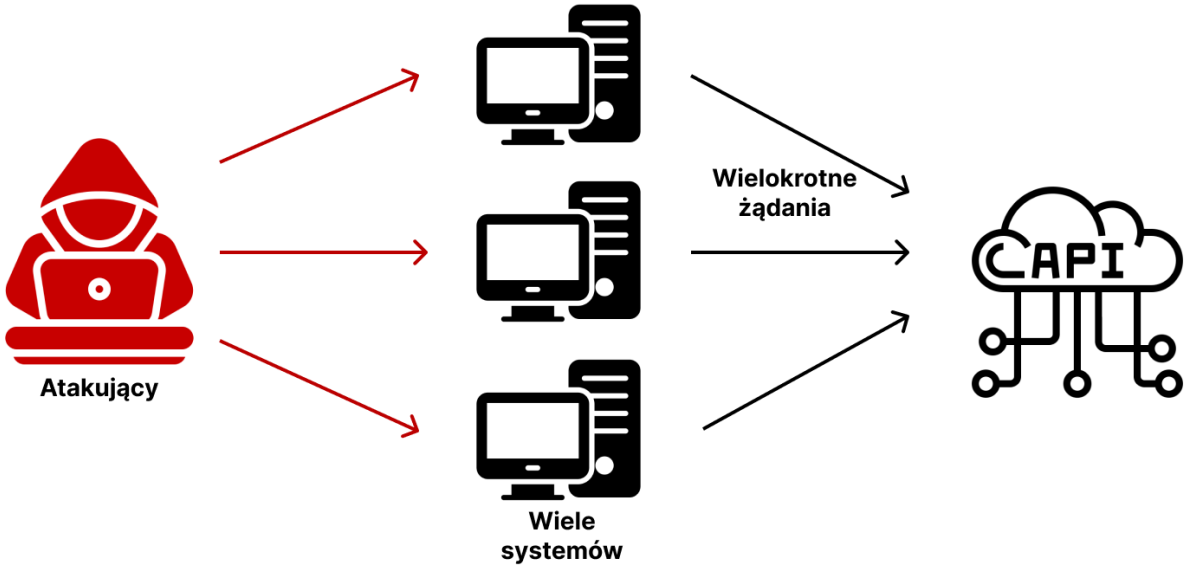
Przedostatnią podatnością i jedną z najniebezpieczniejszych jest *brak limitu zapytań*. Jeśli w aplikacji z interfejsem API nie jest wprowadzony limit zapytań może być on powodem braku dostępności serwera ze względu na przeciążenia nadmiernymi żądaniami. Fakt ten może prowadzić do ostatniej podatności jaką jest atak *blokady usług* (ang. Denial of Service).

Atak DoS [5] mogą wykorzystywać podatność braku limitu zapytań. Celem takiego ataku jest zablokowanie dostępu do usług serwerowych za pomocą wysyłania do serwera wielu pakietów, aby wykorzystać jego cały zasobów obliczeniowy. Dodatkowo często wykorzystywany jest on do zapełnienia miejsca na dyskach zazwyczaj poprzez wysyłanie masywnych plików do serwisów FTP. Na Rys 6 zaprezentowany został przykładowy schemat przebiegu ataku typu blokady usługi.



Rys 6 Przykładowy schemat ataku blokady usług – opracowanie własne

Istnieje również inna odmiana DDoS, czyli *rozproszona odmowa usługi* (ang. Distributed Denial of Service) [5]. Jej cel jest taki sam jak w przypadku DoS, natomiast różni się jego sposób realizowania, ponieważ w tym przypadku zamiast jednej stacji roboczej wykorzystywane jest wiele komputerów w tym samym czasie. Ten typ DoS jest wydajniejszy ze względu na wykorzystywanie większej ilości zasobów sprzętowych. Za pomocą Rys 7 został zaprezentowany przykładowy schemat realizacji ataku rozproszonej odmowy usługi.



Rys 7 Przykładowy schemat ataku rozproszonej odmowy usługi – opracowanie własne

3.2. Przykłady incydentów związanych z lukami w bezpieczeństwie

Aby zobrazować jak kluczowe ma znaczenie odpowiedniego zabezpieczenia aplikacji przed niepożądanym działaniem osób trzecich w tym podrozdziale zostaną zaprezentowane przykłady incydentów cybernetycznych jakie miały miejsce na przestrzeni ostatnich lat.

3.2.1. Zbyt duża ekspozycja danych w serwisie Twitter

Pierwszy z incydentów dotyczył bardzo popularnego serwisu społecznościowego Twitter [6], który świadczy usługi mikrobloga. Atak ten miał miejsce najprawdopodobniej na przełomie roku 2021 i 2022. Do jego realizacji hakerzy posłużyli się luką one-day, czyli podatnością, która istnieje od zerowego dnia wykrycia, to oznacza, że cyberprzestępcy znajdują taką lukę i wykorzystują ją zanim zostaną wdrożone jakiekolwiek środki zaradcze.

Luka polegała na zbyt dużej ekspozycji danych podczas próby logowania, umożliwiło to atakującym zidentyfikowanie oraz wykradzenie między innymi takich informacji jak numer telefonu i adres e-mail dla ponad pięciu milionów kont użytkowników serwisu, należy jednak podkreślić, że podczas tego wycieku nie ujawniono, żadnych haseł użytkowników.

Podatność została wdrożona w jednej z aktualizacji serwisu w czerwcu 2021 roku, natomiast została ona wykryta dopiero w styczniu 2022 roku, za pomocą programu oferowanego przez Twittera, w którym można badać zabezpieczenia serwisu, a następnie zgłaszać znajdowane luki. Dodatkowo serwis zniwelował tę lukę dopiero w okolicach lipca 2022 roku, kiedy skradzione dane zostały wystawione na sprzedaż oraz ze względu na to, że

nie posiadali wcześniej informacji o tym, aby luka została wykorzystana przez cyberprzestępców.

Twitter poinformował swoich użytkowników o takim incydencie i doradził, aby nie dołączali adresu e-mail oraz numerów telefonów do swoich kont. Ponadto, aby zmaksymalizować bezpieczeństwo swoich użytkowników podpowiedział, żeby każdy zastosował na swoim koncie uwierzytelnienie dwuetapowe.

3.2.2. Wyciek danych klientów T-Mobile

Kolejnym incydentem związanym bezpośrednio z API dotyczy jednej z wiodących marek świadczących usługi telekomunikacyjne – T-Mobile. Był to już ósmy atak cybernetyczny na firmę T-Mobile od roku 2018. Cyberprzestępca rozpoczął atak na jeden z wielu interfejsów API T-Mobile w dniu 25 listopada 2022, dzięki czemu uzyskał on do niego dostęp i mógł wykradać dane osobowe klientów. Niestety operator nie ujawnił informacji, w jaki sposób doszło do ataku, natomiast można się domyślić, że osoba atakująca mogła znaleźć lukę w interfejsie API i ją wykorzystać do swoich celów. W dniu 5 stycznia 2023 roku, T-Mobile wykrył podejrzane czynności i od razu postanowił zapobiec działaniom atakującego poprzez usunięcie mu dostępu do API w czasie 24 godzin.

Według odpowiedzi T-Mobile, cyberprzestępcy poprzez zaatakowanie API wykradli dane około trzydziestu siedmiu milionów klientów w tym: imię i nazwisko, adres rozliczeniowy, adres e-mail, numer telefonu, datę urodzenia oraz numer konta T-Mobile. Warto podkreślić, że API, do którego się włamano umożliwiało dostęp tylko do podstawowych danych abonentów, w związku z czym wrażliwe dane klientów takie jak: dane kart kredytowych, hasła lub numery rządowe obywatela nie weszły w posiadanie cyberprzestępcy. Niemniej jednak za pomocą wykradzionych danych przestępcy mogą na przykład podszywać się pod klientów T-Mobile, aby pozyskać dodatkowe informację.

W ramach przeciwdziałania cyberprzestępcy, który zaatakował interfejs, T-Mobile postanowił zgłosić naruszenie do amerykańskiej agencji federalnej, aby sprawa została zbadana. Dodatkowo w celu zadbania o swoich klientów, postanowili ostrzec wszystkich, którzy mogli zostać objęci atakiem, poprzez poinformowanie o zaistniałym incydencie. Jedną z ważnych informacji, jest to, że podejrzana aktywność została całkowicie zablokowana oraz nie ma żadnych informacji o tym, że atakujący zdołał zaburzyć działania systemów lub sieci T-Mobile [7].

4. Metodologie testów bezpieczeństwa

Niniejszy rozdział pracy magisterskiej skupia się na kluczowych aspektach związanych z metodologią testów bezpieczeństwa, które stanowią ważną rolę w analizie oraz ocenie podatności infrastruktur informatycznych. Przedstawia jak należy realizować planowanie testów penetracyjnych bazując na typowym ataku cybernetycznym. Dodatkowo omawia narzędzia, z jakich można skorzystać w celu przeprowadzania oceny stanu zabezpieczeń systemu.

4.1. Etapy przeprowadzania testów penetracyjnych

Planowanie testów penetracyjnych stanowi bardzo ważną kwestię w testowaniu bezpieczeństwa infrastruktur informatycznych. Najlepszym podejściem do planowania testów jest postawienie się w roli atakującego, który chce przeprowadzić typowy atak cybernetyczny, dzięki czemu możemy określić szczegółowo cel oraz sposób realizacji testów. Aby zaplanować testy zaprezentowany zostanie, jak przebiega typowy atak cybernetyczny złożony z pięciu kroków [5].

W pierwszym kroku realizowane jest zbieranie informacji o atakowanej infrastrukturze informatycznej. Dla potencjalnego cyberprzestępcy jest to najważniejszy krok w całym procesie atakowania, ze względu na to, że pozwala określić ścisły proces przeprowadzania ataku. Atakujący zazwyczaj polegają na dwóch podejściach do zbierania informacji. Pierwszą z nich jest rekonesans pasywny [5], który polega na gromadzeniu wszelkich informacji o atakowanej infrastrukturze za pomocą ogólnodostępnych źródeł takich jak na przykład serwisy społecznościowe. Ta metoda zbierania informacji jest bardzo trudna lub nawet niemożliwa do wykrycia. Drugą metodą jest rekonesans aktywny [5], jest on znacznie podatniejszy na wykrycie z uwagi na interakcje cyberprzestępcy z atakowaną infrastrukturą informatyczną, ponieważ dotyczy to na przykład skanowanie portów czy też poszukiwania podatności w systemie za pomocą pasywnych skanów.

Kolejnym krokiem po rozeznaniu jest określenie lub zrealizowanie konkretnego skryptu bądź narzędzia, które zostanie wykorzystane do ataku i dostarczenie go do ofiary. Sam wybór narzędzia oraz metoda dostarczenia w dużej mierze zależy od samego atakującego. Przykładowo, cyberprzestępca może wykorzystać pocztę elektroniczną do przesłania zainfekowanego pliku jako załącznika w wiadomości e-mail.

Trzecim krokiem jest sam atak za pomocą odpowiedniego exploitu [5]. Dzięki czemu cyberprzestępca może osiągnąć zamierzone działania. W tym kroku cyberprzestępca może na przykład posłużyć się danymi do kont, które zostały wykradzione z innych systemów i wykorzystać je w celu zrealizowania ataku metodą siłową.

Następne etapy, czyli czwarty oraz piąty są ze sobą powiązane, ponieważ dotyczą one uzyskania uprawnień do atakowanej infrastruktury. W czwartym kroku cyberprzestępcy próbują przełamać zabezpieczenia w celu zmaksymalizowania uprawnień do poziomu administratora systemu, przy czym, starają się również pozyskać dostęp do wielu kont w oprogramowaniu. Natomiast w kroku piątym, atakujący starają się utrzymać dostęp do atakowanego systemy, dzięki któremu będą mieli zawsze do niego dostęp.

Na podstawie przedstawionego typowego ataku cybernetycznego możemy teraz sformułować plan testów penetracyjnych. Ze względu na to, że badania będą dotyczyły testowego API skupimy się na kluczowych trzech punktach, które umożliwią wykrycie podatności. W pierwszym etapie zostanie przeprowadzone zbieranie informacji o atakowanym API. Natomiast z uwagi na to, że jest to testowe oprogramowanie mamy wszelkie informacje o testowym środowisku. Dlatego zostaną przedstawione narzędzia z jakich możemy skorzystać w celu przeprowadzenia dogłębnego rekonesansu API. W kolejnym kroku planowania skupimy się na przeanalizowaniu zabezpieczeń jakie na chwilę obecną zostały wdrożone do testowego serwera opartego na interfejsie API. W ostatnim kroku zostaną wykorzystane między innymi gotowe exploity w celu zasymulowania ataku i wyszukaniu podatności. Wyróżnione trzy kroki zostaną przedstawione w rozdziale szóstym dotyczącym testom bezpieczeństwa.

4.2. Narzędzia do testowania bezpieczeństwa

Podrozdział ten poświęcony jest do przedstawienia narzędzi jakimi możemy się posłużyć w celu identyfikacji podatności aplikacji czy też systemów informatycznych. W naszym przypadku będą one służyć do przetestowania interfejsu API pod względem bezpieczeństwa. Dodatkowo zaprezentowane zostaną oprogramowania przeznaczone do zbierania informacji o interfejsie API.

4.2.1. Kali Linux

Pierwszym podstawowym narzędziem jest system operacyjny *Kali Linux* [5] opierający się na dystrybucji Debian. Jest to jedna z popularnych dystrybucji Linuxa, która cechuję się rozwiązaniem typu - otwartego kodu źródłowego zapoczątkowanym przez Mati Aharoni oraz Devon Keares. Przeznaczony głównie do wspierania między innymi testów penetracyjnych, ale

również jest szeroko stosowany w informatyce śledczej oraz do łamania zabezpieczeń infrastruktur informatycznych.

W aktualnej wersji Kali Linuxa możemy znaleźć wiele narzędzi, około pięciuset o rozbudowanych funkcjonalnościach przeznaczonych do realizowania na przykład testów penetracyjnych, wykonywania audytów bezpieczeństwa czy do przeprowadzania szczegółowych analiz śledczych.

Z naszego punktu widzenia Kali Linux zostanie wykorzystany do przeprowadzenia testów bezpieczeństwa, wykorzystując jego niektóre z wielu narzędzi wspomagających proces testowania. Jednym z takich narzędzi, które dostarcza system to Burp Suit, który przedstawiony jest w następnej kolejności.

4.2.2. Burp Suit

Burp Suite [9] jest to jedno z najlepszych narzędzi, które oferuje szeroki wachlarz funkcji do testowania bezpieczeństwa infrastruktur informatycznych znajdujących się w sieci. To najczęściej wykorzystywane oprogramowanie do badania bezpieczeństwa i testowania luk w oprogramowaniu. Składa się on z kilku głównych modułów, które umożliwiają testowanie bezpieczeństwa oraz ręczne znajdywanie podatności.

Pierwszym z nich jest wykorzystywanie proxy, dzięki czemu możliwe jest przechwytywanie, podglądanie, a nawet modyfikowanie żądania HTTP przesyłanego na przykład poprzez przeglądarkę internetową do serwerów.

Kolejnym modułem jest skaner umożliwiający automatyczne skanowanie infrastruktury informatycznej w celu znajdowania podatności. Niestety w wersji darmowej Burp Suit skaner nie jest dostępny. Wykrywanie luk realizowane jest za pomocą wysyłania wielu żądań do serwera, a następnie skaner analizuję odpowiedzi i stwierdza czy dana aplikacja posiada daną lukę. Skanery tego typu posiadają jedną wadę, która została opisana we wcześniejszym rozdziale drugim.

Ostatnim kluczowym modułem jest pająk. Jego zadaniem jest symulowanie zachowania człowieka na stronach internetowych, dzięki czemu automatycznie przegląda aplikację internetowe poprzez odwiedzanie wszystkich odnośników oraz na ich podstawie mapuję strukturę witryny internetowej, dzięki której możemy uzyskać wszystkie punkty końcowe, do których odwołuję się witryna internetowa.

Podsumowując Burp Suite jest bardzo potężnym narzędziem służącym do identyfikacji oraz eliminacji potencjalnych luk w zabezpieczeniach infrastruktur informatycznych przed atakami cyberprzestępców. Dodatkowo, dzięki jego możliwościom konfiguracyjnym oraz elastyczności jest szeroko stosowany przez specjalistów do spraw cyberbezpieczeństwa.

4.2.3. OWASP ZAP

Kolejnym kluczowym narzędziem w testowaniu bezpieczeństwa aplikacji internetowych jest *OWASP ZAP* [10]. To rozwiązanie o otwartym kodzie źródłowym przeznaczone do między innymi testów penetracyjnych. Jej działanie jest podobne do Burp Suit, czyli również wykorzystuję proxy, dzięki czemu działa pomiędzy serwerem, a aplikacją odpowiadająca za interakcję z użytkownikiem. Jest to pewnego rodzaju oprogramowanie działające podobnie do ataku „człowiek pośrodku", czyli przechwytywaniu oraz modyfikacji żądań HTTP. OWASP ZAP może również pochwalić się automatycznym skanerem, jednak w tym narzędziu jest on całkowicie darmowy w odróżnieniu do Burp Suite.

Narzędzie to dysponuje również pająkiem, dzięki któremu możemy analizować witrynę internetową. Natomiast w OWASP ZAP, aby to rozwiązanie było bardziej skuteczne zalecane jest korzystanie z niego razem z ręczną eksploracją witryny. W ten sposób zmapujemy większość aplikacji oraz punktów końcowych do których się odwołuje i która może posiadać potencjalne luki w zabezpieczeniach.

W naszym badaniu wykorzystamy między innymi darmowy skaner z OWASP ZAP, za pomocą którego przeskanujemy interfejs API i otrzymamy listę podatności wraz z ich priorytetami. Będzie to podstawa do zrealizowania testów penetracyjnych, które sprawdzą wykryte podatności.

4.2.4. Postman

Również bardzo ważnym narzędziem jest *Postman* [11]. Jest to narzędzie nie związane bezpośrednio z testowaniem bezpieczeństwa, natomiast pozwala odwoływać się do interfejsów API za pomocą żądań HTTP wykorzystując różne metody tego protokołu (np.: POST, GET). Rozwiązanie to przeznaczone jest do testowania, budowania, dokumentowania oraz udostępniania żądań API. Postman cieszy się dużą popularnością wśród programistów tworzących serwery oparte o interfejs API, ze względu na jego szeroki wachlarz funkcjonalny.

Jedną z najważniejszych funkcji jakie oferuje Postman jest wykonywanie testów jednostkowych dla żądań HTTP. Dzięki czemu umożliwia automatyczne sprawdzanie odpowiedzi z serwera, które możemy analizować i reagować na nie w określony sposób.

Postman w naszym badaniu zostanie wykorzystany do zmapowania interfejsu API do dwóch poprzednich narzędzi. Dodatkowo za jego pomocą możemy w prosty sposób sprawdzić czy zaprogramowany interfejs API posiada ograniczenie dostępu do zasobów dla nieuprawnionych użytkowników.

4.2.5. Wireshark

Ostatnim narzędziem, które zostanie wykorzystane do testowania zabezpieczeń testowego API jest *Wireshark* [2]. Jest to narzędzie stosowane głównie do analizy ruchu sieciowego, dzięki przechwytywaniu i analizowaniu pakietów przesyłanych w sieci lokalnej oraz w całym Internecie. Ze względu na licencję otwartego kodu źródłowego na której jest udostępniany jego społeczność stale dodaje nowe funkcjonalności do programu. Oprócz podsłuchiwania oraz skanowania sieci umożliwia między innymi odzyskiwanie haseł Wi-Fi, rejestrowanie danych pakietów oraz ich eksportowanie, filtrowanie ruchu sieciowego, a nawet generowanie statystyk i raportów odnoszących się do przechwyconych pakietów danych.

Oprogramowanie Wireshark zostanie użyte w celu określenia bezpieczeństwa w kontekście próby przechwycenia żądania, na przykład do punktu końcowego uwierzytelniania. Dzięki przechwyceniu tego żądania będzie można między innymi określić czy testowy interfejs API korzysta z zabezpieczonej warstwy transportowej w komunikacji internetowej, czy też przesyła dane w sposób jawny.

4.2.6. Kiterunner

Pierwszym z narzędzi przeznaczonych do zbierania informacji bezpośrednio zrealizowane z myślą o API jest *Kiterunner* [14]. Jest to oprogramowanie, które umożliwia wykrywanie punktów końcowych bazując między innymi na zbiorze potencjalnych słów jakie mogą się w nich znajdować. Dodatkowo narzędzie jest wyposażone w ogromną ilość danych z plików opisujących różne interfejsy API, bo aż około sześciuset tysięcy, które udoskonalają wyszukiwanie punktów końcowych. Posiada również bardzo cenną funkcjonalność jaką jest powtarzanie zapytań do punktów końcowych, które wcześniej zostały wykryte przez program. Jest to bardzo potężne narzędzie, ponieważ umożliwia wysyłanie około trzydziestu tysięcy żądań na sekundę, dzięki którym skutecznie poszukuję punktów końcowych API.

4.2.7. OWASP Amass

Kolejne narzędzie to *Amass* [15]. Jest to narzędzie z rodziny *OWASP*, które również umożliwia przeprowadzenie rekonesansu o interfejsie API. Amass realizuje proces zbierania informacji za pomocą aktywnego oraz pasywnego rekonesansu. Odbywa się to analogicznie do opisywanego w rozdziale 4 typowego ataku cybernetycznego, gdzie cyberprzestępca wykorzystuję dwie metody rekonesansu. Natomiast OWASP Amass oba rozeznania wykonuje w sposób automatyczny szukając informacji o API w ogólnodostępnych źródłach internetowych oraz za pomocą aktywnych skanów interfejsu. Jedyne co należy dostarczyć do programu to adres do interfejsu API, a narzędzie przeskanuje je pod kątem między innymi punktów końcowych.

Część Badawcza

5. Przygotowanie wirtualnego środowiska testowego

Niniejszy rozdział piąty rozpocznie część badawczą nad bezpieczeństwem testowego interfejsu API. W każdym badaniu bezpieczeństwa infrastruktur informatycznych kluczowym elementem jest zadbanie o odpowiednie środowisko przeznaczone do testów, które będzie odseparowane od środowiska przeznaczonego dla użytkowników końcowych ze względu na szkody jakie mogą zostać wyrządzone podczas testowania bezpieczeństwa. Z uwagi na to rozdział ten jest przeznaczony do omówienia jak możemy przygotować środowisko testowe przeznaczone do badań nad bezpieczeństwem interfejsu API. Ponadto przedstawia jak wygląda architektura zaimplementowanego API oraz omawia konfigurację narzędzi przeznaczonych do testów bezpieczeństwa.

W naszym przypadku w pierwszej kolejności należy wdrożyć system Kali Linux na wirtualną maszynę *VirtualBox* [12], opracowane oprogramowanie przez firmę Oracle. Dzięki zastosowaniu wirtualizacji możemy stworzyć osobny system gościa, który będzie oddzielony od systemu hosta. Na tak przygotowane środowisko wirtualne możemy następnie lokalnie wdrożyć testowe API, a dzięki narzędziom, które dostarcza Kali Linux dokonać testów bezpieczeństwa.

5.1. Architektura Aplikacji Spring Boot

Realizowane API testowe dotyczy tematyki transportowej oraz zrealizowane jest we wcześniej wymienionej technologii Spring Boot. Dzięki wykorzystaniu właśnie tego frameworka otrzymujemy oprogramowanie, które możemy wdrażać na wszystkich systemach posiadających wirtualne środowisko uruchomieniowe Java w wersji 8. Głównym założeniem API jest usprawnianie działania firmy transportowej pod kątem zarządzania nią oraz jej pracownikami i kierowcami. Zaprogramowane API na chwilę obecną umożliwią logowanie, dodawanie, usuwanie, modyfikowanie użytkowników i pracowników oraz ich stanowisk. Dodatkowo posiada możliwość tworzenia pokoi rozmów dla pracowników, którzy są użytkownikami systemu. Chat pomiędzy pracownikami zrealizowany jest za pomocą WebSocketów.

W testowym API zostały wdrożone trzy rolę, które mogą posiadać użytkownicy. Za ich pomocą możliwe jest ograniczenie dostępu do punktów końcowych, natomiast szczegółowo zostanie to opisane w kolejnym rozdziale odnośnie do analizowania zabezpieczeń testowego interfejsu API. Pierwszą z ról jest „`Admin`” jest to najwyższy możliwy stopień dostępu, ponieważ za jego pomocą możemy odwoływać się do wszystkich zasobów API. Dodatkowo

tylko ta rola upoważniona jest do zakładania kont nowych użytkowników. Kolejną rolą jest „`Operator`”, który przypisywany jest dla pracowników biurowych firmy takich jak na przykład spedytorzy. Ostatnią rola to „`Driver`”, czyli kierowca i ona została wdrożona z myślą o kierowcach ciężarówek. Wszystkie punkty końcowe wraz z opisem i rolą jakie mogą się do niej odwoływać zostały zaprezentowane za pomocą Tabela 1.

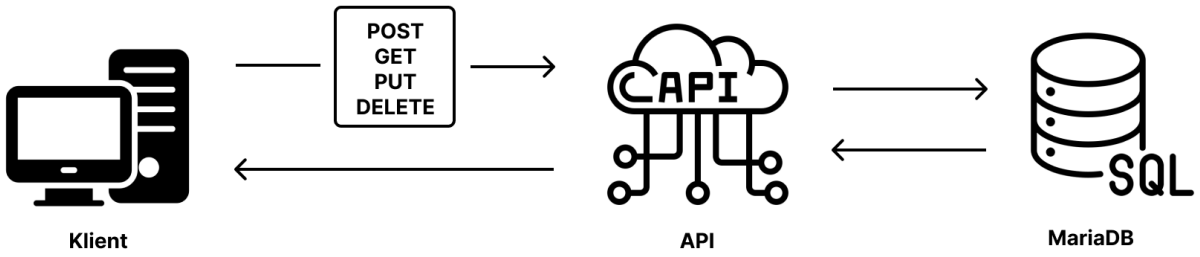
Punkt końcowy	Rola dostępu	Opis
/api/auth/authenticate	Ogólnodostępne	Punkt końcowy służący do uwierzytelniania użytkowników
/api/auth/refreshToken/{refreshToken}	Ogólnodostępne	Punkt końcowy służący do odświeżania JWT, jeżeli wygaśnie oraz jeśli sesja nie wygasła.
/api/sign-out	Ogólnodostępne	Punkt końcowy służący do wylogowania użytkownika
/api/v1/users/readUserDetails/{userId}	Administrator	Punkt końcowy służący do pobierania szczegółowych informacji o użytkowniku
/api/v1/users/readAllUsersDetails	Administrator	Punkt końcowy, który zwraca listę szczegółowych informacji o użytkownikach. Umożliwia stronicowanie.
/api/v1/users/readBasicUserDetails	Administrator Operator Driver	Punkt końcowy pobierający podstawowe dane o zalogowanym użytkowniku
/api/v1/users/readLoggedUserInformation/{userId}	Administrator Operator Driver	Punkt końcowy pobierający szczegółowe informację o zalogowanym użytkowniku

Punkt końcowy	Rola dostępu	Opis
/api/v1/users/createuser	Administrator	Punkt końcowy służący do utworzenia konta dla pracownika
/api/v1/users/updateAvatar/{userId}	Administrator Operator Driver	Punkt końcowy służący do aktualizowania zdjęcia użytkownika
/api/v1/users/updateUser/{userId}	Administrator	Punkt końcowy służący do aktualizowania informacji o użytkowniku
/api/v1/users/deleteUser/{userId}	Administrator	Punkt końcowy, dzięki któremu można usunąć konkretnego użytkownika
/api/v1/employee/addEmployee	Administrator Operator	Punkt końcowy służący do utworzenia nowego pracownika w systemie
/api/v1/employee/updateEmployee/{employeeId}	Administrator Operator	Punkt końcowy służący do aktualizacji informacji o pracowniku
/api/v1/employee/getAllEmployeesDetails	Administrator Operator	Punkt końcowy służący do pobierania listy szczegółowych informacji o wszystkich pracownikach. Umożliwia stronicowanie.
/api/v1/employee/getBasicEmployeeInformation	Administrator Operator	Punkt końcowy służący do pobierania listy z podstawowymi informacjami o wszystkich pracownikach. Umożliwia stronicowanie.

Punkt końcowy	Rola dostępu	Opis
/api/v1/employee/getEmployeeDetails/{employeeId}	Administrator Operator	Punkt końcowy służący do pobierania szczegółowych informacji o konkretnym pracowniku
/api/v1/employee/deleteEmployee/{employeeId}	Administrator Operator	Punkt końcowy służący do usuwania pracownika
/api/v1/employee/addJobPosition/{positionName}	Administrator Operator	Punkt końcowy dodający nowe stanowisko w firmie
/api/v1/employee/updateJobPosition/{jobPositionId}/{newPositionName}	Administrator Operator	Punkt końcowy służący do aktualizacji stanowiska
/api/v1/employee/getJobPositions	Administrator Operator	Punkt końcowy, dzięki któremu możemy pobierać listę stanowisk
/api/v1/employee/deleteJobPosition/{jobPositionId}	Administrator Operator	Punkt końcowy odpowiadający za usuwanie stanowiska
/api/v1/users/readAllRoles	Administrator Operator	Punkt końcowy służący do pobierania ról jakie znajdują się w API
/api/v1/message/{roomName}	Administrator Operator Driver	Punkt końcowy odpowiadający za pobieranie listy wiadomości z konkretnego pokoju rozmów
/api/v1/message/create/{roomName}	Administrator Operator Driver	Punkt końcowy, dzięki któremu możemy tworzyć nowe pokoje rozmów między użytkownikami

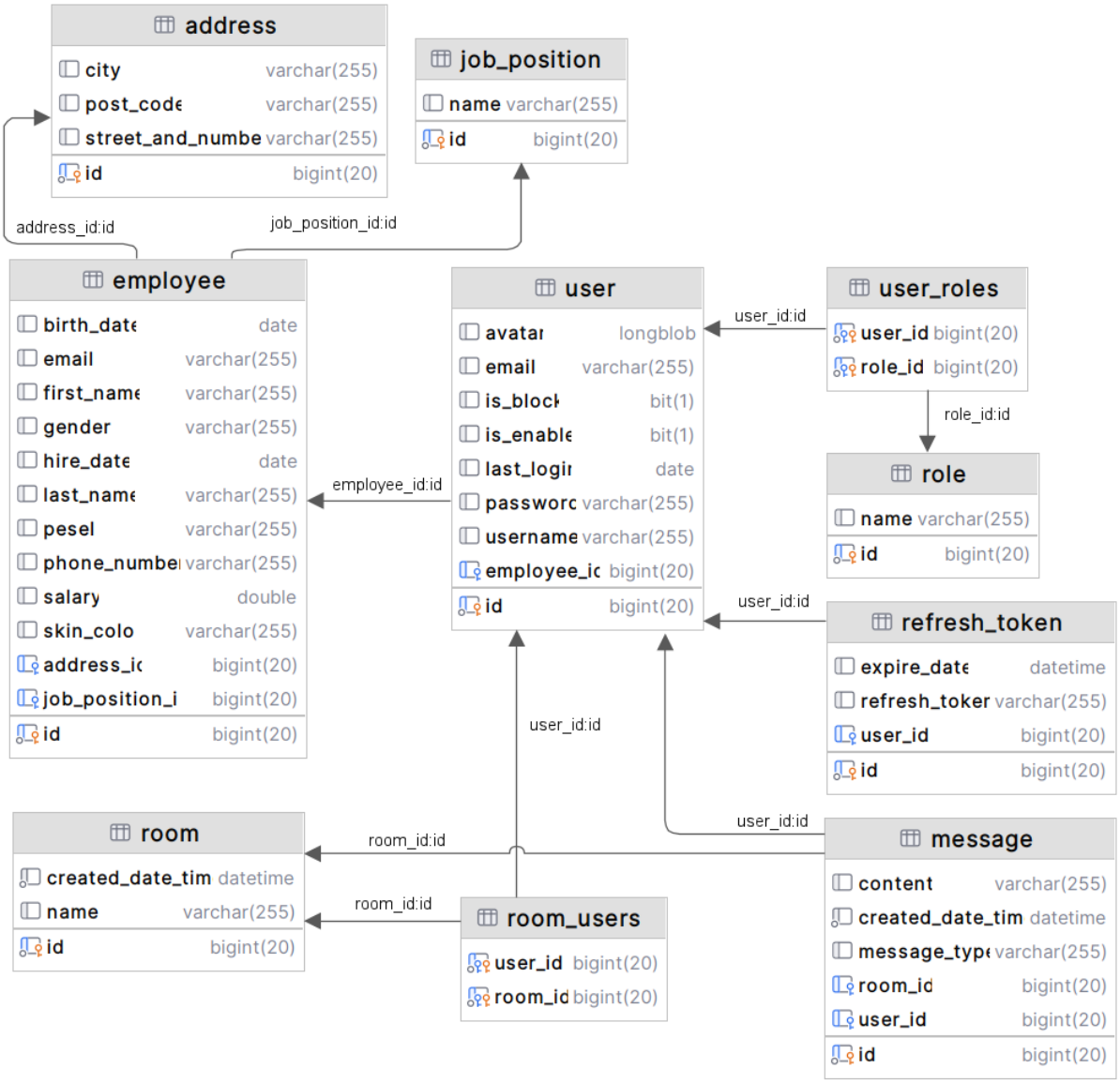
Tabela 1 Spis wszystkich punktów końcowych testowego API

Interfejs API zaprojektowany jest, aby udostępnił dane w zgodzie ze standardem REST, który został szczegółowo opisany w rozdziale drugim. Za pomocą Rys 8 zaprezentowane jest jak przebiega przykładowe odwoływanie się do REST API. Klient wysyła żądania HTTP do API, a jeśli to konieczne API odwołuję się do bazy danych i zwraca dane dla klienta.



Rys 8 Przykładowy schemat odwoływania do REST API – opracowanie własne

Warstwą danych wykorzystywana przez testowy interfejs jest relacyjna baza danych *MariaDB* [13]. Dzięki tej technologii możliwe jest tworzenie struktur danych nazywanych tabelami (Encje) wraz z możliwymi powiązaniami między nimi. Powiązania umożliwiają łatwy i szybki dostęp do danych zawartych w bazie danych. Dodatkowo w testowym interfejsie API został wykorzystany framework Hibernate [4], który implementuje moduł JPA (ang. Java Persistence Api). Zadaniem Hibernate jest mapowanie struktur bazy danych (Encji) na obiekty java i obiekty na struktury bazy danych, co nazywane jest *mapowaniem obiektowo-relacyjnym* (ang. Object-Relational Mapping). Natomiast JPA jest interfejsem, który umożliwia zarządzanie relacyjnymi bazami danych w tym bazą MariaDB przy wykorzystaniu tak zwanych repozytoriów, które dotyczą konkretnego modelu obiektowego. Dzięki wykorzystaniu Hibernate nie stosujemy języka SQL, natomiast możemy posługiwać się „Query Method", czyli zapytaniami generowanymi za pomocą JPA wykorzystując do tego nazwę metody zawartą w interfejsie repozytorium lub z wykorzystaniem HQL (ang. Hibernate Query Language). Za pomocą Rys 9, który przedstawia diagram ERD zaprezentowany został schemat bazy danych oraz powiązania między encjami.



Rys 9 Diagram ERD bazy danych – opracowanie własne

5.2. Wdrożenie Aplikacji Spring Boot na wirtualne środowisko

Prawidłowe działanie aplikacji spring boot API, w pierwszej kolejności wymaga wdrożenia relacyjnej bazę danych MariaDB na zwirtualizowany testowy system Kali Linux, ze względu na wymaganą zależność bazy danych dla API. Aby to zrealizować należy:

- Zaktualizować system Kali Linux - `sudo apt update`
- Zainstalować system bazy danych - `sudo apt install mariadb-server`
- Uruchomić serwis mariadb - `sudo systemctl start mariadb`

Jeżeli operacja instalacji przebiegła pomyślnie możemy przejść do konfigurowania bazy danych. Odpowiednie skonfigurowanie bazy danych pod testowe API jest kluczowe i głównie

polega na stworzeniu bazy danych o nazwie `db` oraz nadaniu hasła dla użytkownika, z uwagi na dane jakie znajdują się w pliku konfiguracyjnym interfejsu API, aby mogło bezproblemowo połączyć się z bazą danych. Aby skonfigurować bazę danych należy:

- Zalogować się na konto administratora - `sudo mysql -u root -p`
- Ustawić nowe hasło „`root`" dla konta administratora - `ALTER USER 'root'@'localhost' IDENTIFIED BY 'root'`
- Stworzenie bazy danych o nazwie „`db`" - `CREATE DATABASE db`

Sukcesywne zainstalowanie oraz skonfigurowanie systemu bazodanowego skracają nam znacznie proces wdrożenia API do środowiska testowego. Wdrożenie podmiotu do badań wymaga sprawdzenia jaka wersja wirtualnej maszyny Javy (ang. Java Virtual Machine) jest zainstalowana na systemie, ponieważ interfejs API wymaga JVM w wersji 1.8. Jeżeli wersja JVM się zgadza możemy przejść do uruchamiania API za pomocą komendy „`java -jar SpringBootApi.jar`". Wykonanie komendy uruchomi serwer API, dzięki czemu mamy gotowe środowisko testowe.

5.3. Konfiguracja narzędzi do testów

Badanie będzie odbywało się za pomocą narzędzi opisanych w poprzednim czwartym rozdziale. W celu skonfigurowania narzędzi Burpsuite oraz OWASP ZAP, aby przechwytywały ruch kierowany do API jedyne co trzeba zrobić to ustawić ich funkcję proxy, które działa jako pośrednik. Do tego celu należy wejść w każdym z narzędzi do ustawień, a następnie proxy i skonfigurować adres na localhost oraz na wolny port na przykład `8085`. Następnie w programie Postman również, należy ustawić proxy na ten sam adres i port, czyli `localhost:8085`. Dzięki temu wszystkie żądania wysyłane za pomocą aplikacji Postman będą kierowane przez narzędzia monitorujące Burpsuite oraz OWASP ZAP, umożliwiając przy tym między innymi pasywne skanowanie API pod kątem podatności czy wykonanie testu sprawdzającego wrażliwość na metodę siłową.

6. Testy bezpieczeństwa

Rozdział szósty skupiony jest na przeprowadzeniu badań dotyczących zabezpieczeń w testowym API. Zostanie zaprezentowane między innymi planowanie testów, w którym omówione jest jakimi narzędziami możemy zbierać informację o infrastrukturze informatycznej. Natomiast głównym celem niniejszego rozdziału jest przeprowadzenie badań za pomocą aktywnego skanowania czy testów sprawdzających między innymi wykryte podatności z przeprowadzonego skanu interfejsu API.

6.1. Planowanie testów

W celu przeprowadzenia dobrze badań nad bezpieczeństwem testowego API, należy zrealizować plan testów. Z uwagi na to skupimy się na trzech ważnych kwestiach podczas planowania. Pierwszą z nich jest zbieranie informacji o testowanym API, następnie zostaną przeanalizowane zabezpieczenia jakie zostały wdrożone do interfejsu, a następnie omówione jest, jak zostaną przeprowadzone zasymulowane ataki na testowy interfejs.

6.1.1. Zbieranie informacji o interfejsie API

Z uwagi na to, że jest to testowe oprogramowanie, które zostało zrealizowane specjalnie do przeprowadzenia badania, posiadamy wszelkie informacje jakie potrzebne są, aby dokonać testów bezpieczeństwa. Natomiast w rozdziale 4 zostały zaprezentowane dwa narzędzia, dzięki którym możemy zbierać informację o interfejsach API za pomocą aktywnego oraz pasywnego rekonesansu. W celu przeprowadzenia badań zostaną wykorzystane informację na temat punktów końcowych, które zaprezentowane są za pomocą Tabela 1.

6.1.2. Analiza zabezpieczeń API

Zrealizowane API testowe posiada jedno główne zabezpieczenie ograniczające dostęp do zasobów dla użytkowników, którzy nie posiadają odpowiednich ról w interfejsie. Całe zabezpieczenie opiera się na *tokenie JWT* (ang. JSON Web Token), który jest tworzony podczas uwierzytelniania użytkownika za pomocą odpowiedniego punktu końcowego. Uzyskany token należy dołączać do nagłówka żądania HTTP za każdym razem, kiedy klient odwołuje się do zabezpieczonych punktów końcowych. Podczas procesu tworzenia tokenu zakodowana w nim jest nazwa użytkownika, dzięki której podczas odwoływania się do zabezpieczonych punktów końcowych będzie ona odczytywana i porównywana czy użytkownik z taką nazwą użytkownika istnieje i czy posiada odpowiednią rolę uprawniającą go do danego zasobu. Jeżeli użytkownik nie posiada odpowiednich uprawnień do zasobu interfejs API zwróci informację o niewystarczających uprawnieniach. Dodatkowo ważność tokenu została ustawiona na czas

trzydziestu sekund, dzięki czemu po upływie tego czasu należy odwołać się do API o odświeżenie tokenu pod specjalnie zrobiony do tego celu punkt końcowy. Wszystkie punkty końcowe API wraz rolą jakie mogą się do nich odwoływać zostały zaprezentowane za pomocą Tabela 1. Dodatkowo do zarządzania bazą danych został wykorzystany framework Hibernate, który sam w sobie posiada zabezpieczenia uniemożliwiające między innymi wstrzykiwanie złośliwego kodu SQL wykorzystując sparametryzowane zapytania.

6.1.3. Omówienie przeprowadzania ataków

Przeprowadzanie kontrolowanych ataków na interfejs API zostanie rozpoczęte od przeskanowania aktywnie aplikacji za pomocą skanera, który znajduję się w narzędziu OWASP ZAP [10]. Dzięki przeskanowaniu interfejsu otrzymamy listę potencjalnych podatności jakie zostały wykryte, które następnie zostaną sprawdzone za pomocą testów manualnych wykorzystując w tym celu narzędzia przedstawione w rozdziale 4. Z wykorzystaniem również testów manualne zostaną sprawdzone zabezpieczenia uwierzytelniania.

6.2. Aktywne skanowanie aplikacji pod kątem podatności

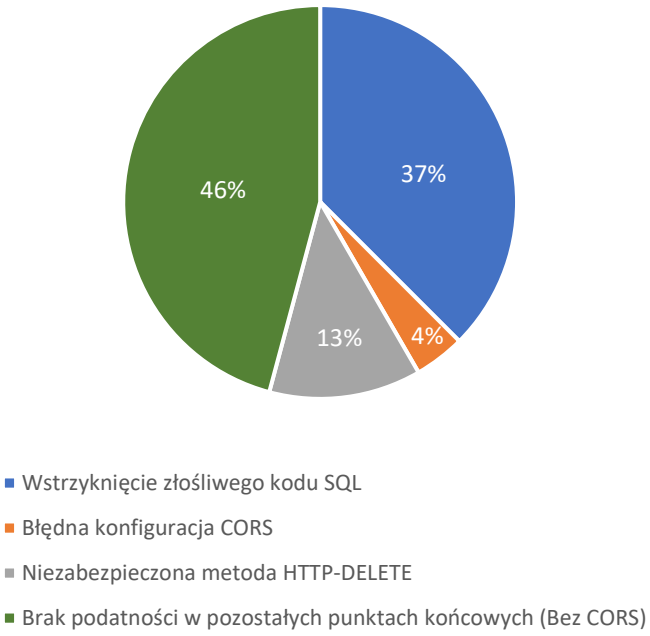
Pasywne skanowanie aplikacji zaczynamy od zmapowania punktów końcowych API do OWASP ZAP za pomocą narzędzia Postman wysyłając żądania do API, które zostają przekierowane za pomocą proxy do narzędzi testujących. W narzędziu OWASP ZAP możemy konfigurować nasilenie skanów, dlatego aby osiągnąć najlepsze wyniki skanowania zrealizowane jest z ustawieniem siły ataku o wartości „*szalony*” (ang. Insane) dla wszystkich możliwych kategorii jakie możemy poddać badaniu. Dodatkowo domyślny próg alarmowania o zagrożeniach został ustawiony na wysoki, dzięki czemu nawet najmniejsze odchylenia sugerujące daną podatność zostaną uwzględnione w liście potencjalnych zagrożeń. Jak wynika z Tabela 2, skanowanie punktów końcowych API wykryło kilka rodzajów podatności w trzynastu punktach końcowych. Najczęściej znajdowaną potencjalną luką w testowym API jest zagrożenie związane ze wstrzykiwaniem złośliwego kodu SQL. Ponadto skanowanie ujawniło takie zagrożenia jak możliwość niezabezpieczonych punktów końcowych realizujących usuwanie danych z systemu za pomocą metody HTTP-DELETE oraz błędną konfigurację mechanizmu CORS (ang. Cross-Origin Resource Sharing).

ID	Punkt końcowy	Podatność	Opis
PK_1	/api/v1/users/readAllUserDetails/	Wstrzyknięcie kodu SQL	Możliwe wstrzykiwanie kodu SQL z wykorzystaniem warunków logicznych
PK_2	/api/v1/employee/updateJobPosition/	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL z wykorzystaniem warunków logicznych do parametru id stanowiska.
PK_3	/api/v1/employee/getAllEmployeeDetails/	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL z wykorzystaniem warunków logicznych 0 AND „1”=”1”
PK_4	/api/v1/employee/updateEmployee/	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL za pomocą warunku logicznego dodanego do nazwiska „Test AND 1=1—„
PK_5	/api/v1/employee/updateJobPosition/	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL za pomocą warunku logicznego dodanego do parametru nazwy stanowiska „NewTest AND 1=2—„
PK_6	/api/v1/users/updateAvatar	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL za pomocą dodania do nazwy pliku „avatar.png ASC –„ jako wartości parametru
PK_7	/api/v1/users/updateUser	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL poprzez dodanie do nazwy użytkownika wartości logicznej „AND 1=1—„

ID	Punkt końcowy	Podatność	Opis
PK_8	/api/v1/employee/addEmployee	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL za pomocą zastosowania warunków logicznych np.: do nazwiska
PK_9	/api/v1/users/createUser	Wstrzyknięcie kodu SQL	Możliwe wstrzyknięcie kodu SQL wykorzystując warunek logiczny np.: w polu hasła
PK_10	Wszystkie	Błędna konfiguracja Cross-Origin Resource Sharing	Wykorzystanie błędnej konfiguracji CORS może pozwolić atakującemu na wykonywanie zapytań AJAX do podatnej witryny ze złośliwej strony załadowanej przez agenta użytkownika ofiary
PK_11	/api/v1/employee/deleteEmployee	Niezabezpieczona metoda HTTP – DELETE	Możliwe niezabezpieczenie metody usuwającej pracownika przed dostępem dla nieuprawnionych osób
PK_12	/api/v1/employee/deleteJobPosition	Niezabezpieczona metoda HTTP – DELETE	Możliwy brak zabezpieczenia metody usuwającej stanowisko przed dostępem dla nieuprawnionych osób
PK_13	/api/v1/users/deleteUser	Niezabezpieczona metoda HTTP – DELETE	Możliwy brak zabezpieczenia metody usuwającej użytkownika przed dostępem dla nieupoważnionych użytkowników

Tabela 2 Spis wykrytych podatności testowego API za pomocą aktywnych skanów – opracowanie własne

Dodatkowo Rys 10 prezentuje wykres ukazujący procentowe zależności do typów podatności jakie zostały wykryte za pomocą aktywnych skanów z OWASP ZAP. Jak można zauważyć, aż trzydzieści siedem procent stanowią zagrożenia związane ze wstrzykiwaniem złośliwego kodu SQL, trzynaście procent odnosi się do możliwego braku w zabezpieczeniu metod DELETE przed dostępem nieuprawnionych użytkowników, natomiast cztery procent związanych jest z brakiem poprawnej konfiguracji mechanizmu udostępniania zasobów miedzy domenami (ang. Cross-Origin Resource Sharing), a czterdzieści siedem procent nie posiada żadnych podatności.



Rys 10 Wykres typów podatności wykrytych podczas aktywnego skanowania testowego API – opracowanie własne

6.3. Testy sprawdzające zabezpieczenia uwierzytelniania

Do sprawdzenia zabezpieczeń uwierzytelniania wykorzystamy dwa testy składające się z próby wykradzenia konta za pomocą zasymulowania ataku metody siłowej oraz sprawdzimy czy możliwe jest zdekodowanie tokenu JSON Web Token, na którym opiera się autoryzacja w testowym API.

6.3.1. Test sprawdzający odporność na metodę siłową

Rozpoczynając od ataku metodą siłową wykorzystamy do tego celu narzędzie OWASP ZAP [10], dzięki któremu możemy dołączyć do danych logowania przesyłanych na odpowiedni

punkt końcowy listę zawierającą najpopularniejsze hasła i nazwy użytkowników. Jeżeli dana infrastruktura nie jest zabezpieczona przed atakami tego typu możemy uzyskać dostęp do kont użytkowników, jeśli wykorzystał takie same dane jakie znajdują się na liście potencjalnych danych logowania. Do przeprowadzenia ataku wykorzystana została lista popularnych danych „rockyou.txt", która posiada około czternastu milionów najczęściej używanych nazw użytkowników oraz haseł.

W testowym API nie zostały wdrożone, żadne elementy ograniczające działania ataku metody siłowej. Ze względu na to zasymulowany atak przebiegł pomyślnie ukazując kolejną podatność badanego podmiotu. Za pomocą zbioru haseł udało się uzyskać dostęp do konta administratora, który posiadał nazwę użytkownika „`admin`" oraz hasło „`password`". Do tego celu OWASP ZAP wykonał ponad piętnaście tysięcy zapytań z różnymi nazwami użytkowników oraz haseł co zaprezentowane jest na Rys 11. Otrzymana odpowiedź o id `15 076` z kodem odpowiedzi `HTTP-200` potwierdza, że atak przebiegł pomyślnie i API zwróciło dane potrzebne do odwoływania się pod zabezpieczone punkty końcowe.

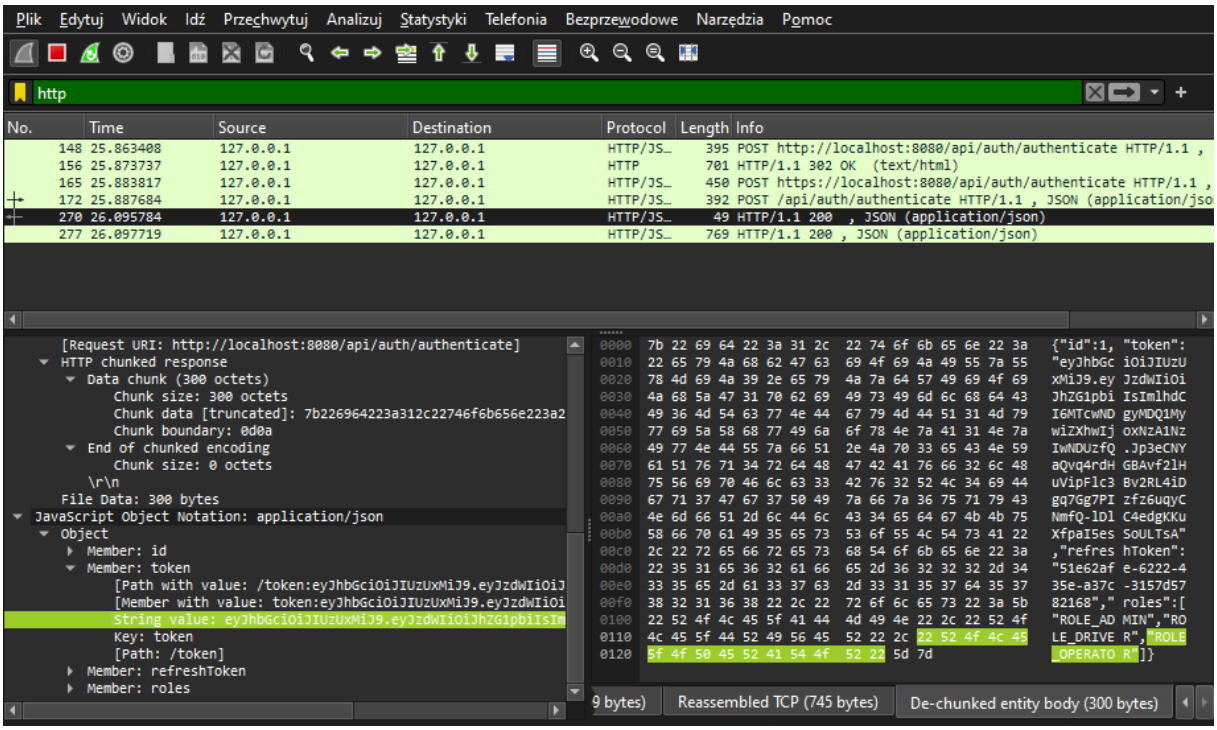
Messages Sent: 16907 Errors: 0 Show Errors

Task ID	Message Ty...	Code	Reason	RTT	Size Resp. H...	Size R...	...	...	Payloads
15 076	Fuzzed	200		630 ms	425 bytes	300 by...			admin, password
3 902	Fuzzed	400		5 ms	396 bytes	112 by...			admin, contraseña
8 611	Fuzzed	400		4 ms	396 bytes	112 by...			admin, muñeca
3	Fuzzed	401		138 ms	424 bytes	97 byt...			admin, 12345

Rys 11 Zrzut ekranu z OWASP ZAP – Pozytywnie zasymulowany atak metodą siłową – opracowanie własne

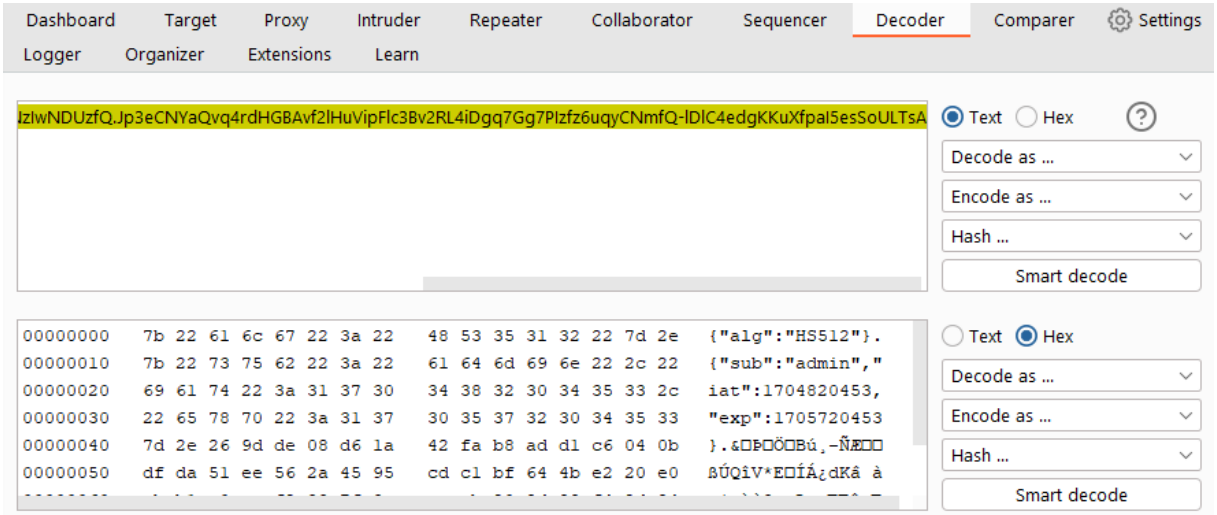
6.3.2. Dekodowanie JSON Web Token

W celu dekodowania tokenu w pierwszej kolejności posłużymy się narzędziem Wireshark [5], dzięki któremu będziemy mogli zasymulować przechwycenie żądania zawierającego odpowiedź z JWT, który następnie poddamy próbie rozkodowania za pomocą narzędzia Burpsuite. W tym celu narzędzie Wireshark ustawiamy, aby przechwytywał ruch w sieci lokalnej z filtrem HTTP, dzięki czemu możemy przechwycić wysyłane żądanie o uwierzytelnienie do API oraz jego odpowiedź co zaprezentowane jest na Rys 12.



Rys 12 Zrzut ekranu z Wireshark – Przechwycenie żądania uwierzytelniania – opracowanie własne

W celu dekodowania przechwyconego tokenu posłużymy się narzędziem Burpsuit. Posiada on wbudowany dekoder dla kodowania typu Base64Url [1]. Jest to kodowanie polegające na przedstawieniu ciągu znaków za pomocą formatu tekstowego wykorzystując do tego zestaw sześćdziesięciu czterech znaków, w którym kombinację są przedstawione jako dane binarne. Dekodując token możemy się dowiedzieć jakie dane są w nim zawarte. Za pomocą Rys 13 przedstawione jest dekodowanie JWT zwracanego z API. Dzięki temu możemy pozyskać informację o czasie ważności tokenu, nazwę użytkownika oraz przy wykorzystaniu jakiego algorytmu został podpisany, w tym przypadku jest to algorytm HS512. Wykonując takie działanie otrzymujemy kolejną potencjalną lukę, w której podpisany token może zostać w łatwy sposób rozkodowany, dzięki czemu atakujący wejdzie w posiadanie nazwy użytkownika. Natomiast nie może on wykorzystać pozyskanego tokenu ze względu na czas wygaśnięcia oraz nie ma możliwości zmodyfikowania zawartości, ponieważ serwer weryfikuję tokeny na podstawię podpisu i jeżeli cyberprzestępca go zmodyfikuje serwer to wykryje i wróci informację o braku uprawnień do żądanego zasobu.



Rys 13 Zrzut ekranu z Burpsuite – rozszyfrowane JWT – opracowanie własne

Dodatkowo za pomocą narzędzia Wireshark ukazała się kolejna podatność jaką jest niezabezpieczony protokół HTTP pomiędzy klientem a serwerem, co oznacza, że jeżeli cyberprzestępca przechwyci żądanie będzie miał dostęp do wszystkich danych, które wysyła klient oraz tych zwracanych przez interfejs API.

6.4. Testy sprawdzające odporność na ataki wstrzykiwania złośliwego kodu SQL

W celu sprawdzenia wykrytych podatności wstrzykiwania zależności jakie zostały wykryte za pomocą aktywnego skanowania zostanie wykorzystane narzędzie Burpsuit [9]. Używając funkcjonalności jaką oferuje w postaci *powtarzacza* (ang. Repeater) umożliwia powtarzanie żądań do API, w których zostanie wstrzyknięty złośliwy kod SQL. Wykorzystując to narzędzie będzie można śledzić, jak zachowuję się interfejs API i określić, czy w danym punkcie końcowych faktycznie istnieje podatność wykryta za pomocą aktywnych skanów czy są to fałszywie pozytywne wyniki. Zagadnienie o fałszywie pozytywnych wynikach zostało omówione w drugim rozdziale pracy.

ID podatności	Wstrzyknięte wartości	Wynik testu
PK_1	• DROP TABLE users; • AND 4201=7661 • AND 1126=1126	Wynik testu ukazał, że dla punktu końcowego skanowanie dało fałszywie pozytywny wynik. Wstrzyknięte wartości nie spowodowały żadnych szkód na serwerze i żądanie zwróciło

ID podatności	Wstrzyknięte wartości	Wynik testu
		wartości z oryginalnego zapytania, pomijając złośliwy kod SQL.
PK_2	• AND 4964=1888-- • OR 1=1--	Wynik testu wykazał, że dla skan również w tym przypadku dał wynik fałszywie pozytywny. Wstrzyknięte wartości zostały wykryte przez serwer zwracając informację o błędnym żądaniu oraz informując w logach o wystąpieniu w żądaniu złośliwych danych
PK_3	• OR 1=1-- • AND 1=1--	Wynik manualnego testu również wykazał, że jest to fałszywie pozytywny wynik, ze względu na poprawne zwrócenie wartości dla oryginalnego zapytania bez brania pod uwagę złośliwego kodu SQL
PK_4	• AND 1=1--	Wynik testu negatywny, co oznacza kolejny fałszywie pozytywny wynik skanu. Natomiast kod, który został wstrzyknięty do wartości klucza w JSON został zapisywany do bazy danych.
PK_5	• AND 1=1--	Wynik testu również negatywny, kolejny fałszywie pozytywny wynik skanu. Serwer po otrzymaniu danych ze złośliwym kodem SQL ignoruje je, natomiast również zapisuje do bazy danych.
PK_6	• ASC – • DROP DATABASE • AND 1=1--	Wynik testu negatywny, serwer ignoruje wstrzyknięte wartości i pozytywnie zapisuje obrazek do bazy danych.

ID podatności	Wstrzyknięte wartości	Wynik testu
PK_7	• AND 1=1— • DROP DATABASE	Wynik testu negatywny, natomiast serwer zapisał złośliwy kod SQL do bazy danych przekazany jako wartość klucza w JSON.
PK_8	• AND 1=1— • DROP DATABASE	Wynik testu negatywny, co daje kolejny fałszywie pozytywny wynik skanowania. Natomiast serwer zapisuję wartości wraz ze złośliwym kodem SQL do bazy danych.
PK_9	• AND 1=1— • DROP DATABASE;	Wynik testu negatywny co oznacza następny fałszywie pozytywny wynik skanowania. Jednakże wstrzyknięta wartość została zapisana do bazy danych.

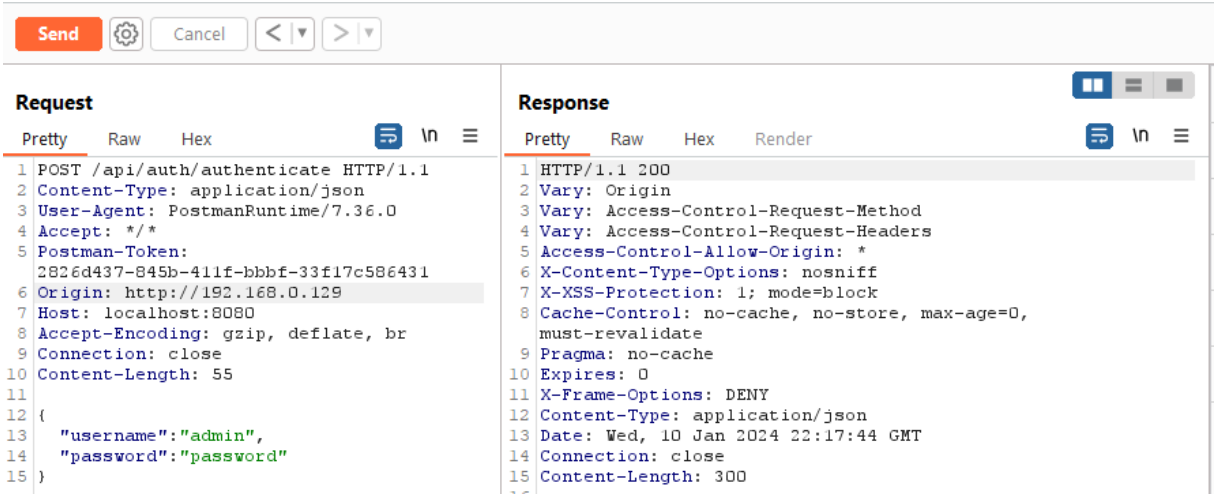
Tabela 3 Testy manualne sprawdzające wykryte podatności wstrzykiwania złośliwego kodu SQL w skanowaniu interfejsu – opracowanie własne

Jak wynika z Tabela 3 wszystkie znalezione podatności wstrzykiwania złośliwego kodu SQL za pomocą aktywnego skanowania okazały się wynikami fałszywie pozytywnymi. Może być to związane z wykorzystanym frameworkiem Hibernate, który posługuję się parametryzowanymi zapytaniami (Query Method, HQL), dzięki czemu przekazywane wartości nie są traktowane jako część kodu SQL, a jak zwykłe dane. Wiąże się to z możliwością zapisania złośliwego kodu SQL jako normalne dane tekstowe w bazie pod zmapowaną zmienną obiektu, a następnie takie dane mogą zostać zwrócone przez API [3].

6.5. Test sprawdzające mechanizm współdzielenia zasobów między domenami

Przeprowadzając test sprawdzający czy wykryta podatność odnosząca się do błędnej konfiguracji mechanizmu CORS nie jest fałszywie pozytywna wykorzystamy ponownie funkcjonalność powtarzacza znajdującego się w narzędziu Burpsuit. Do tego celu przechwycimy żądanie i dodamy nagłówek „`Origin`”, z adresem ip komputera znajdującego się w sieci lokalnej. Na załączonym Rys 14, został zaprezentowany zrzut ekranu z narzędzia Burpsuite. Jak wynika z otrzymanej odpowiedzi serwera w nagłówku zawarta jest linijka:

`Access-Control-Allow-Origin: *`, co oznacza, że interfejs API umożliwia dostęp do zasobów ze wszystkich domen.



Rys 14 Zrzut ekranu z Burpsuite - Test sprawdzający mechanizm CORS – opracowanie własne

W tym przypadku skanowanie OWASP ZAP z sukcesem wykryło podatność, ze względu na fakt, iż taka konfiguracja mechanizmu CORS może być wykorzystywana jedynie w fazie implementowania interfejsu w środowisku developerskim. Wykorzystanie takich ustawień dla aplikacji, która została wdrożona do środowiska produkcyjnego może wiązać się z ryzykiem dostępu do zasobów z niezaufanych źródeł.

6.6. Testy sprawdzające kontrolę dostępu do zasobów

Do zrealizowania testów sprawdzających kontrolę dostępu do zasobów wykorzystane zostanie narzędzie Postman, dzięki któremu możliwe jest odwoływanie się do zasobów, przy jednoczesnym wykorzystaniu tokenu użytkownika, który nie posiada uprawnień do odwoływania się pod dany punktu końcowy.

ID Podatności	Punkt końcowy	Rola dostępu	Wynik testu
PK_11	.../deleteEmploee	Administrator Operator	Wynikiem testu jest ograniczenie dostępu do danego zasobu wyłączenie dla zalogowanych użytkowników posiadających rolę Administratora lub Operator.

PK_12	.../deleteJobPosition	Administrator Operator	Wynikiem testu jest ograniczenie dostępu do danego zasobu wyłączenie dla zalogowanych użytkowników posiadających rolę Administratora lub Operator.
PK_13	.../deleteUser	Administrator	Wynikiem testu jest ograniczenie dostępu do danego zasobu wyłączenie dla zalogowanych użytkowników posiadających rolę Administratora.

Tabela 4 Testy manualne sprawdzające wykryte podatności niezabezpieczonych metod HTTP – DELETE – opracowanie własne

Jak wynika z Tabela 4 wszystkie trzy przypadki wykryte odnoszące się do niezabezpieczonych metod HTTP – DELETE były wynikami fałszywie pozytywnymi. Podczas testów dla niezalogowanych użytkowników serwer zwracał informację ze statusem „`401 Unathorized`" informującym o możliwości dostępu jedynie po zalogowaniu. Natomiast dla użytkowników, którzy są zalogowani, ale nie posiadają odpowiedniej roli, serwer zwracał informację ze statusem „`403 Forbidden`" świadczącym o odmowie dostępu ze względu na zbyt niskie uprawnienia użytkownika. Dodatkowo w ten sam sposób zostały przetestowane wszystkie pozostałe punkty końcowe API pod względem dostępu do nich przez nieuprawnionych i niezalogowanych użytkowników. Wynikiem przeprowadzonych testów, wszystkie z nich posiadają odpowiednie zabezpieczenia umożliwiające dostęp do nich jedynie użytkownikom zalogowanym oraz posiadającym odpowiednią rolę. Punkty końcowe wraz z rolą dostępu zostały zaprezentowane za pomocą Tabela 1 w rozdziale piątym dotyczącym architektury aplikacji Spring Boot API.

6.7. Testy wydajnościowe podczas ataku metodą siłową

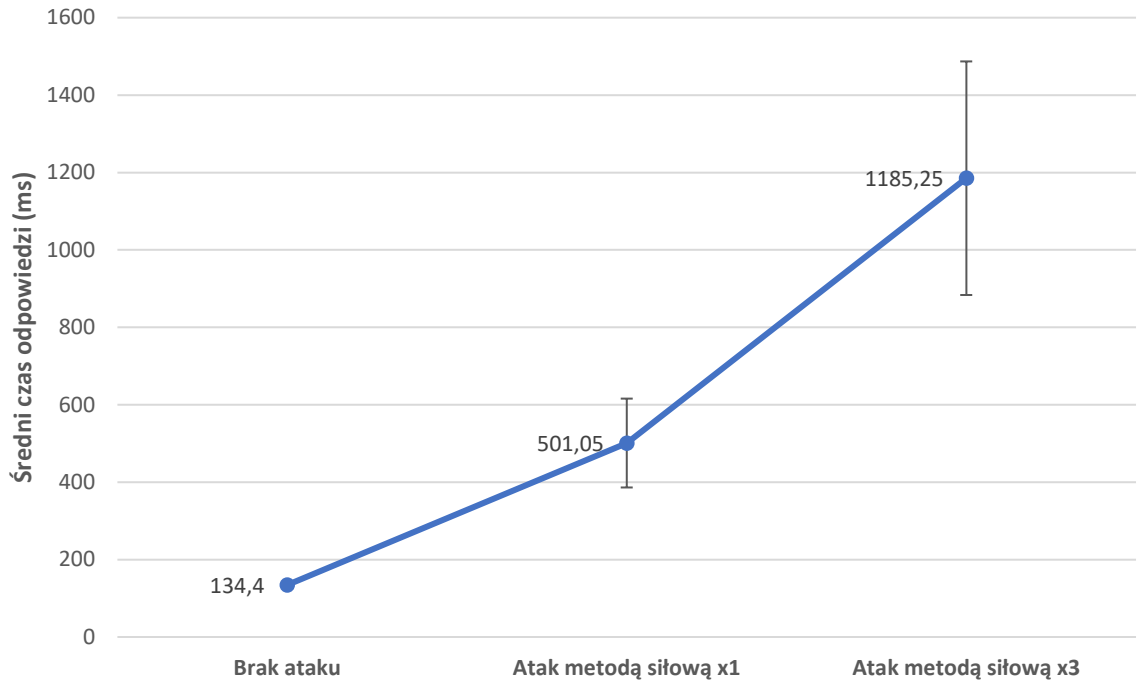
Realizacja testów wydajnościowych podczas ataku wymaga zasymulowania ataku, który będzie obciążał interfejs API. Do tego celu zostanie wykorzystana wykryta podatność na atak metodą siłową. Główną miarą sprawdzenia wydajności API będzie czas odpowiedzi na żądania pod wybrane trzy punkty końcowe. Zostaną przeprowadzone trzy testy pomiaru czasu. Pierwszy z nich będzie dotyczył czasu odpowiedzi API, jeżeli nie będzie w jego kierunku żadnych ataków. Drugi, jeżeli w czasie żądania będzie przeprowadzany jeden atak metodą

siłową oraz trzeci, jeśli w kierunku API będą skierowane trzy ataki metodą siłową w tym samym czasie. Należy jednak pamiętać, że testy przeprowadzane są w środowisku wirtualnym, który posiada ograniczenia wydajności. Za pomocą Tabela 5 zaprezentowane zostały czasy odpowiedzi na żądania pod punkt końcowy uwierzytelniania użytkownika dla trzech badanych sytuacji.

Brak ataku (ms)	**Atak metodą siłową x1 (ms)**	**Atak metodą siłową x3 (ms)**
132	351	901
139	346	612
128	401	934
136	592	1300
146	664	1080
126	458	1009
131	326	961
138	704	1412
134	487	1376
139	568	1130
129	642	1015
130	417	1366
135	653	1144
134	430	1060
139	531	1131
128	574	1515
127	542	1171
137	564	1437
147	393	2120
133	378	1031
Średnia		
134,4	501,05	1185,25
Odchylenie standardowe średniej		
5,7218878	114,78174	301,73215

Tabela 5 Czasy odpowiedzi na żądania dla punktu końcowego uwierzytelniania – opracowanie własne

Na podstawie danych z Tabela 5 został sporządzony wykres liniowy przedstawiający wzrost średniego czasu odpowiedzi na żądania podczas braku ataku, jednego ataku oraz trzech ataków metodą siłową. Wykres został zademonstrowany za pomocą Rys 15.



Rys 15 Wzrost średniego czasu odpowiedzi punktu końcowego uwierzytelnia – opracowanie własne

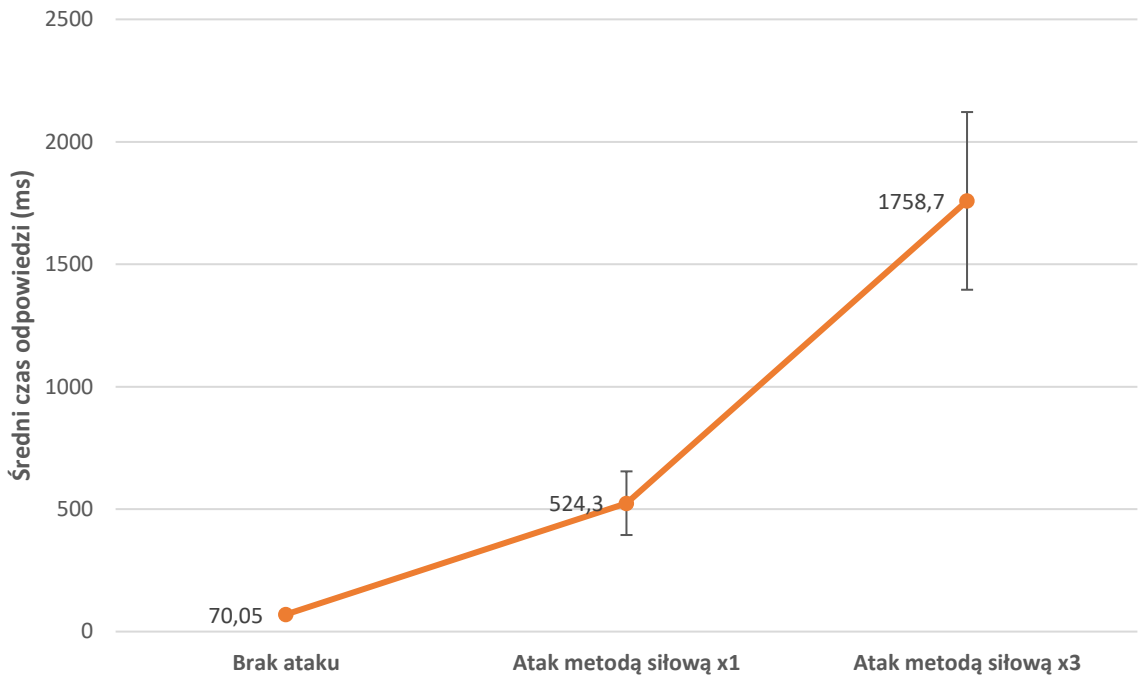
Kolejnym zbadanym punktem końcowym pod kątem czasu odpowiedzi jest ścieżka umożliwiająca pobieranie wszystkich szczegółowych danych na temat użytkowników. Dane przedstawione w Tabela 6 pokazują czasy odpowiedzi API dla trzech badanych scenariuszy.

Brak ataku (ms)	Atak metodą siłową x1 (ms)	Atak metodą siłową x3 (ms)
100	557	1514
65	675	1638
61	543	1820
88	650	1795
84	424	1068
71	511	1635
70	536	1053
74	551	1637
58	792	2140
54	488	1862
76	676	1719

Brak ataku (ms)	Atak metodą siłową x1 (ms)	Atak metodą siłową x3 (ms)
58	341	1975
65	571	1000
72	264	2170
58	549	1716
64	460	2030
57	437	2222
67	293	2113
79	499	1996
80	669	2071
Średnia		
70,05	524,5	1758,7
Odchylenie standardowe średniej		
11,599461	129,99812	362,78961

Tabela 6 Czasy odpowiedzi na żądania dla punktu końcowego pobierającego szczegółowe dane o wszystkich użytkownikach – opracowanie własne

Z wykorzystaniem informacji z Tabela 6 zaprezentowany został wykres liniowy za pomocą Rys 16, który obrazuje wzrost średniego czasu odpowiedzi dla badanego punktu końcowego.



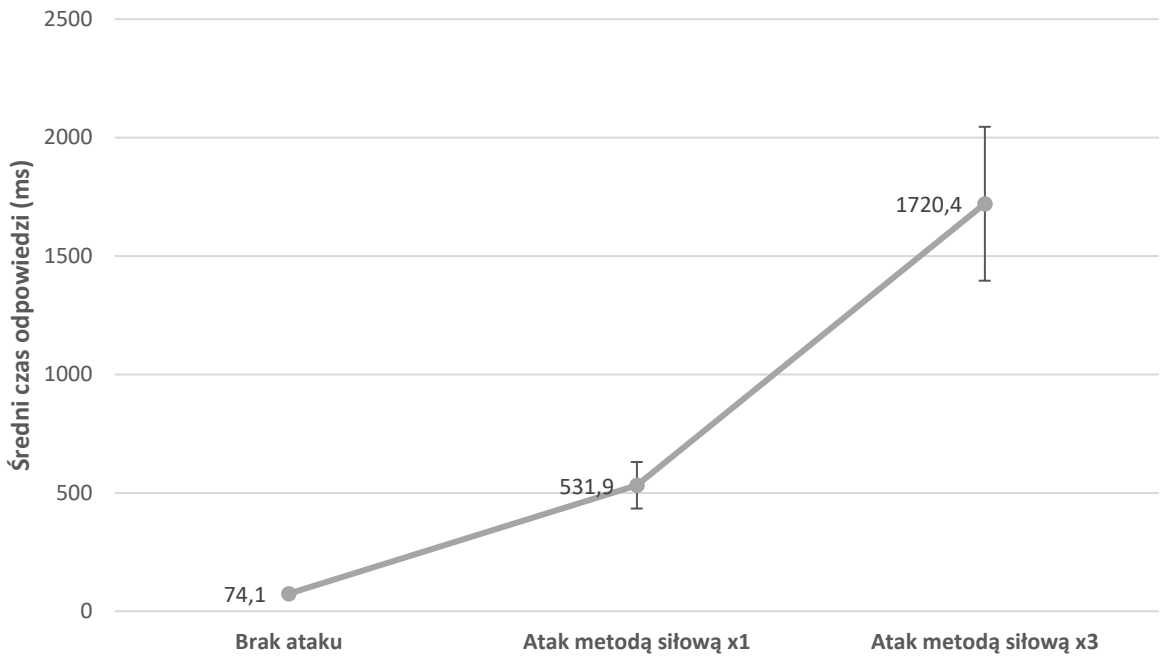
Rys 16 Wzrost średniego czasu odpowiedzi punktu końcowego pobierającego szczegółowe dane użytkowników- opracowanie własne

Ostatnim punktem końcowym poddanym testowi wydajności jest zasób umożliwiający pobieranie szczegółowych informacji na temat wszystkich pracowników. W oparciu o Tabela 7 zawierającą czasy odpowiedzi, stworzono wykres liniowy obrazujący średni wzrost czasu odpowiedzi API. Wykres zaprezentowany jest za pomocą Rys 17.

Brak ataku (ms)	Atak metodą siłową x1 (ms)	Atak metodą siłową x3 (ms)
80	477	1354
68	577	1967
75	510	1845
82	642	1168
81	584	1629
88	601	1853
83	592	1754
80	334	2000
54	565	1721
88	592	1703
82	432	1667
68	547	1746

78	370	1903
62	585	2043
62	748	1650
59	475	2531
85	438	1609
58	449	1664
71	644	1700
78	476	901
Średnia		
74,1	531,9	1720,4
Odchylenie standardowe średniej		
10,309704	98,280161	324,93452

Tabela 7 Czasy odpowiedzi na żądania dla punktu końcowego pobierającego szczegółowe dane o wszystkich pracownikach – opracowanie własne



Rys 17 Wzrost średniego czasu odpowiedzi punktu końcowego pobierającego szczegółowe informacje o wszystkich pracownikach – opracowanie własne

Analizując zebrane dane dotyczące czasów odpowiedzi dla trzech scenariuszy oraz zrealizowane na ich podstawie wykresy można wyciągnąć wnioski, iż API osiągnęło bardzo dobre średnie wyniki czasu odpowiedzi podczas braku ataków w trzech badanych punktach

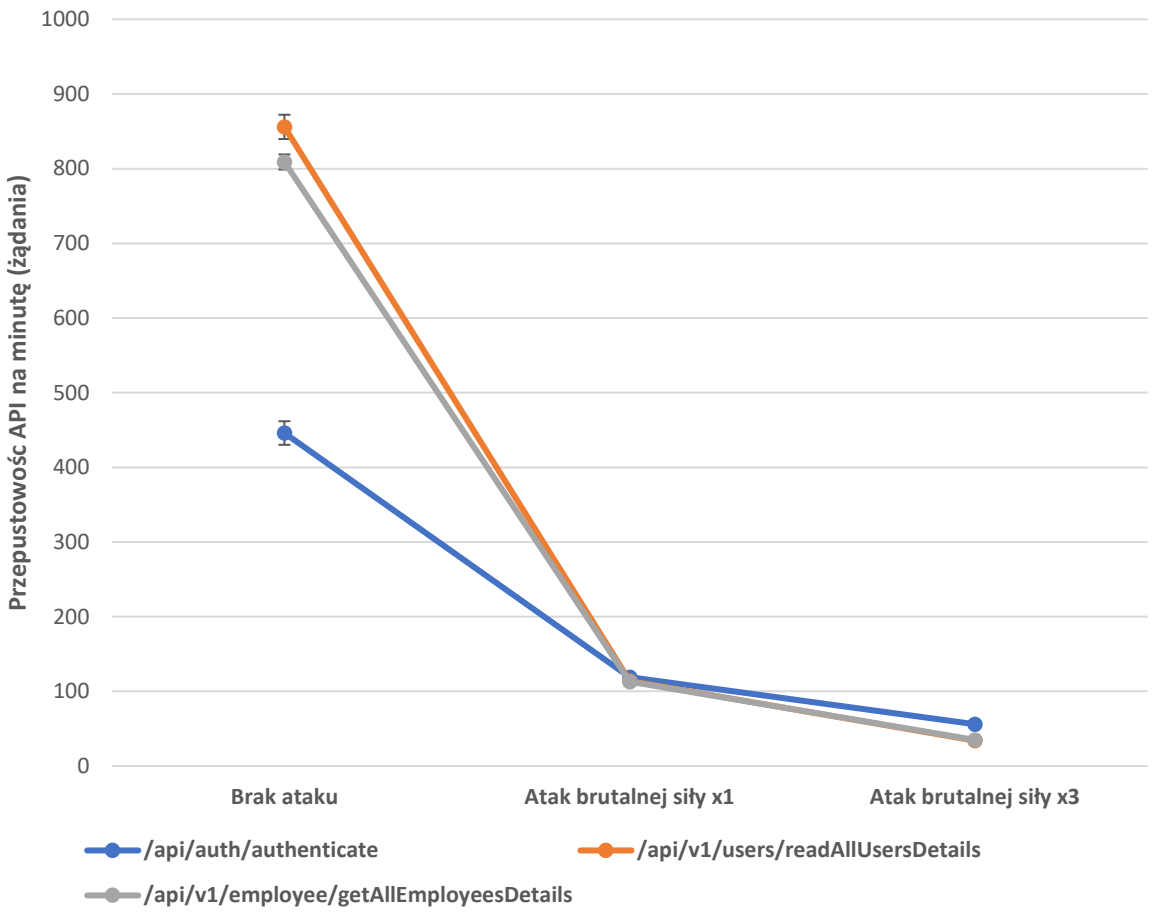
końcowych, co jest pozytywnym osiągnięciem dla normalnych warunków pracy interfejsu. W przypadku scenariusza wystąpienia jednego ataku metody siłowej średni czas odpowiedzi API oscylował w okolicach pięciuset milisekund, co oznacza wzrost w porównaniu do braku ataku, natomiast nie jest on drastyczny i w dalszym ciągu umożliwia efektywne korzystanie z interfejsu. Natomiast podczas sytuacji potrójnego atakowania API, jego wydajność znacznie zmalała w porównaniu z brakiem ataku oraz pojedynczym atakiem, co może wiązać się ze słabą przepustowością interfejsu.

Wykorzystując średnie czasy odpowiedzi możemy dokonać obliczeń dzieląc minutę przez średni czas odpowiedzi, dzięki czemu otrzymamy potencjalną przepustowość API dla każdego punktu końcowego uwzględniając trzy badane scenariusze. Przepustowość interfejsu została zaprezentowana za pomocą Tabela 8, obrazującej średnią liczbę żądań jakie obsługuje API w jednostce czasu wynoszącą jedną minutę.

Punkt końcowy	Brak ataku	Atak metodą siłową x1	Atak metodą siłową x3
/api/auth/authenticate	446	119	56
/api/v1/users/readAllUsersDetails	856	114	34
/api/v1/employee/getAllEmployees Details	809	113	35

Tabela 8 Potencjalna przepustowość API wybranych punktów końcowych dla każdego scenariusza – opracowanie własne

Z wyników przedstawionych w Tabela 8, można zauważyć wyraźną tendencję spadku przepustowości API wraz ze zwiększającą się liczbą ataków metody siłowej. W przypadku mniejszej przepustowości interfejsu dla punktu końcowego uwierzytelniania podczas braku ataku w porównaniu do pozostałych dwóch punktów końcowych może być to spowodowane większą złożonością logiki jakie musi wykonać serwer w celu sprawdzenia tożsamości użytkownika. Spadek przepustowości został również zobrazowany za pomocą wykresu liniowego znajdującego się na Rys 18, zrealizowanego na podstawie danych z Tabela 8.



Rys 18 Spadek przepustowości interfejsu w zależności od liczby ataków – opracowanie własne

Biorąc pod uwagę czasy odpowiedzi i przepustowość interfejsu w zależności od ilości ataków można stwierdzić, że istotną kwestią poprawiającą efektywne korzystanie z API jest wdrażanie zabezpieczeń uniemożliwiające ataki, które w stanowczy sposób obciążają interfejs (np.: metoda siłowa, rozproszona odmowa usługi).

7. Podsumowanie wyników testów

Pomyślne zbadanie interfejsu API za pomocą testów bezpieczeństwa, umożliwia klasyfikację oraz priorytetyzację znalezionych podatności w zależności od potencjalnych szkód jakie mogą wyrządzić. W tym celu niniejszy rozdział poświęcony jest między innymi na przedstawienie wykrytych podatności wraz z ich sklasyfikowaniem oraz określeniem w jakiej kolejności należy wdrażać zabezpieczenia przeciwko nim. W fazie badania bezpieczeństwa za pomocą testów wykryto następujące podatności:

- Brak wdrożonego protokołu HTTPS
- Wstrzykiwanie złośliwego kodu SQL – Brak walidacji danych wejściowych
- Atak metodą siłową
- Błędna konfiguracja mechanizmu CORS
- Brak zaszyfrowania danych w tokenie JWT

7.1. Klasyfikacja i priorytetyzacja znalezionych podatności

W celu klasyfikacji oraz priorytetyzacji znalezionych podatności wykorzystana została skala określająca wpływ jakie niesie ze sobą dana podatność na bezpieczeństwo API. Składa się ona z trzech poziomów: niski, średni oraz wysoki. Stopień z oznaczeniem niski, otrzymają podatności, które stanowią niewielkie zagrożenie w kontekście bezpieczeństwa oraz powinny zostać niwelowane jako ostatnie. Średni stopień jest przeznaczony dla luk, które mogą zostać wykorzystane do ataków oraz które są w stanie spowodować większe zagrożenie w odróżnieniu do niskich. Ostatnim wysokim stopniem są oznaczone zagrożenia, które wymagają natychmiastowej uwagi i to nimi należy zająć się w pierwszej kolejności, ze względu na najwyższe potencjalne ryzyko wykorzystania w ataku cybernetycznym. Za pomocą Tabela 9 została zaprezentowana klasyfikacja podatności w zależności od zagrożenia bezpieczeństwa.

Podatność	**Skala ryzyka**
Brak wdrożonego protokołu HTTPS	Wysoki
Atak metodą siłową	Wysoki
Wstrzykiwanie złośliwego kodu SQL – Brak walidacji danych wejściowych	Średni
Brak zaszyfrowania danych w tokenie JWT	Średni
Błędna konfiguracja mechanizmu CORS	Niski

Tabela 9 Klasyfikacja oraz priorytetyzacja podatności – opracowanie własne

Jak wynika z Tabela 9, aż dwie wykryte podatności posiadają klasyfikację na poziomie wysokim, dwie o poziomie średnim oraz jedna o skali niski. Pozwala nam to określić, które aspekty bezpieczeństwa należy wdrażać do testowego interfejsu w pierwszej kolejności. Brak wdrożonego protokołu HTTPS posiada skalę wysoki, ze względu na niezabezpieczoną komunikację pomiędzy klientem, a serwerem. Podatność może prowadzić do przechwycenia żądania przez cyberprzestępcę, który jest w stanie na przykład z łatwością wejść w posiadanie poufnych danych. Ze względu na sukcesywnie zasymulowany atak metody siłowej otrzymał skalę wysoki, ponieważ istnieje realne zagrożenie wykradzenia kont użytkowników. W przypadku braku walidacji danych wejściowych skala posiada wartość średni, ze względu na zapisywanie wstrzykniętego kodu sql jako zwykłej wartości w bazie danych. Kolejną podatnością jest brak zaszyfrowania danych w tokenie JWT. Luka otrzymała skalę średni ze względu na możliwość przechwycenia żądania i pozyskania nazwy użytkownika, która może posłużyć między innymi do ataku metodą siłową. Ostatnią podatnością ze skalą niski jest błędna konfiguracja mechanizmu CORS, która może umożliwiać atakującemu odwoływanie się do interfejsu z innych domen.

7.2. Rekomendacja i środki zaradcze

W oparciu o wykryte podatności w niniejszym podrozdziale zostaną zaprezentowane rekomendację oraz środki zaradcze jak należy zabezpieczać interfejs przed atakami.

W pierwszej kolejności należy zabezpieczyć komunikację pomiędzy klientem, a serwerem wdrażając zabezpieczenie dla warstwy transportowej komunikacji internetowej. Można to zrealizować za pomocą wykorzystania protokołu HTTPS [1], dzięki któremu otrzymujemy zaszyfrowane połączenie. Protokół ten do zabezpieczenia warstwy transportowej wykorzystuję protokół *bezpiecznych warstw gniazd* (ang. Secure Sockets Layer) lub jego następcę *bezpieczeństwo warstwy transportowej* (ang. Transport Layer Security), dzięki którym otrzymujemy zabezpieczenie danych przed przechwyceniem, odczytaniem lub modyfikacją. Zaszyfrowanie danych utrudnia cyberprzestępcy jakiekolwiek działania z przechwyconymi informacjami.

Ważnym aspektem bezpieczeństwa jakie należy wdrażać do interfejsów API, aby przeciwdziałać atakom, które obciążają serwer (np.: atak metody siłowej lub blokady usług) to limitowanie zapytań [1]. Sukcesywne wdrożenie limitu zapytań oscyluje w okolicach od stu do pięciuset żądań do API w ciągu minuty. Natomiast punkt końcowy uwierzytelniania powinien

mieć własny limit, który umożliwia od trzech do pięciu prób logowania w ciągu piętnastu minut.

Kwestia związana z walidacją danych powinna być zawsze uwzględniana i nie powinno się ufać klientom czy też bezpośrednio powiązanym aplikacjom, które odwołują się do API, ponieważ one również mogą w niewłaściwy sposób walidować przesyłane informację lub zostać wykorzystane między innymi do ataku typu wstrzykiwania złośliwego kod SQL lub zapisany atak skryptów między witrynami. Z tego względu należy wdrażać walidację wszystkich danych przesyłanych na serwer pod kątem przede wszystkim składności oraz niepożądanych znaków [1].

Aspekt bezpieczeństwa związany z brakiem szyfrowania danych w tokenie jest również istotny z uwagi na łatwość zdekodowania tokenu i pozyskania danych w nim zawartych. Ze względu na to wszystkie dane jakie znajdują się w tokenach należy szyfrować. W przypadku testowego API jedynie serwer wykorzystuje dane zawarte w tokenie dlatego do zabezpieczenia informacji można wykorzystać szyfrowanie symetryczne oparte o jeden klucz szyfrujący i deszyfrujący, dzięki któremu jedynie serwer będzie mógł operować na informacjach zawartych w JWT [1].

Ostatnia rekomendacja dotyczy mechanizmu udostępniania danych pomiędzy domenami. Serwery, które udostępniają zasoby, zawsze powinny wdrażać mechanizm CORS, ze względu na możliwość ograniczania domen, które powinny mieć dostęp do interfejsu. Wdrożony oraz dobrze skonfigurowany mechanizm CORS, umożliwia serwerowi decydowanie czy klient należy do zaufanych domen, którym należy udostępniać zasoby [1].

8. Doskonalenie bezpieczeństwa aplikacji Spring Boot

Niniejszy rozdział pracy poświęcony jest na wdrożenie zabezpieczeń bazując na wykrytych podatnościach oraz rekomendacjach, które zostały przedstawione w poprzednich częściach pracy. Dodatkowo po wdrożonych zabezpieczeniach interfejs API zostanie poddany ponownemu aktywnemu skanowaniu, w celu sprawdzenia czy podatności zostały z sukcesem wyeliminowane.

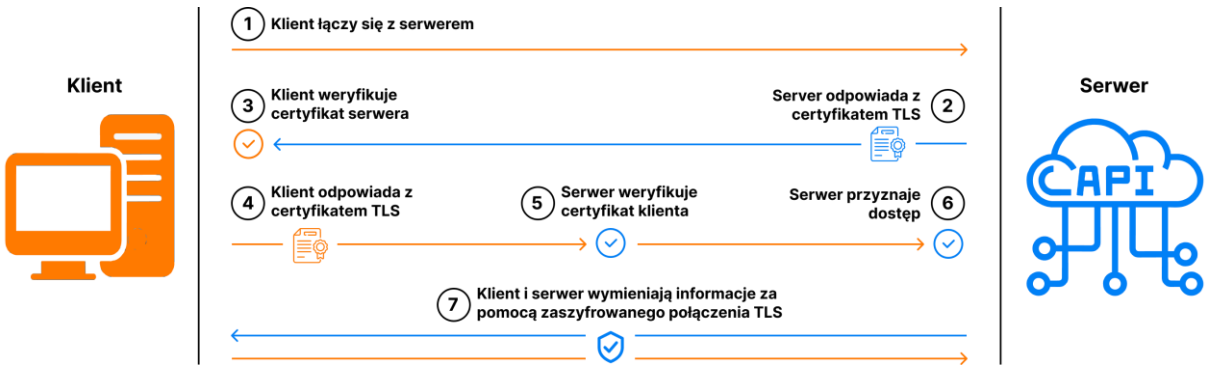
8.1. Poprawki i modyfikacje w aplikacji

Doskonalenie interfejsu należy realizować w kolejności zależącej od skali ryzyka danej podatności, która została wykryta. Z uwagi na to, kolejne punkty doskonalenia API zostały zrealizowane na podstawie Tabela 9.

8.1.1. Wdrożenie protokołu HTTPS

W pierwszej kolejności należy wdrożyć zabezpieczony protokół HTTPS. Do tego celu zostało wykorzystane narzędzie OpenSSL [16] oraz narzędzie keytool [17] wchodzące w skład Javy. OpenSSL to biblioteka dostarczająca szereg funkcji przeznaczonych do szyfrowania, uwierzytelniania oraz zabezpieczania komunikacji internetowej za pomocą między innymi generowania certyfikatów. Z kolei keytool, to narzędzie służące do zarządzania magazynami kluczy oraz certyfikatami.

Wdrożone zabezpieczenie do interfejsu opiera się na wzajemnym TLS (ang. Mutual TLS) [16]. Dzięki takiemu rozwiązaniu klient jak i serwer muszą się wzajemnie zweryfikować, aby możliwe było utworzenie bezpiecznego połączenia. Weryfikacja opiera się na przesłaniu przez klienta jak i serwera certyfikatu, który umożliwia zweryfikowanie tożsamości drugiej strony. Z wykorzystaniem Rys 19, został zaprezentowany przykładowy schemat obrazujący kroki jakie są realizowane w celu nawiązania bezpiecznego połączenia.



Rys 19 Schemat nawiązywania połączenia za pomocą dwustronnego TLS – opracowanie własne

Na podstawie przedstawionego schematu to klient inicjalizuję chęć nawiązania połączenia, na który serwer odpowiada przesyłając klientowi certyfikat serwera, w którym znajduje się klucz publiczny wraz z niezbędnymi informacjami do weryfikacji przez klienta otrzymanego certyfikatu. Następnie po weryfikacji klient przesyła swój certyfikat wraz kluczem publicznym i informacjami do zweryfikowania certyfikatu przez serwer. Jeżeli wymienione certyfikaty zostały pomyślnie zweryfikowane ustanawiane jest zabezpieczone połączenie, dzięki któremu odbywa się przesyłanie informacji.

Wykorzystując wcześniej wymienione narzędzia został zrealizowany certyfikat symulujący główny urząd certyfikatów, który umożliwi bezpośrednie podpisanie certyfikatu klienta oraz serwera. Następnie zostały utworzone certyfikaty przeznaczone dla interfejsu API oraz dla klienta, które zostały podpisane zasymulowanym certyfikatem. Kolejnym krokiem było wyeksportowanie certyfikatu serwera oraz jego klucza prywatnego do magazynu kluczy w formacie PKCS12, który może zostać użyty w aplikacji Spring Boot. W ostatnim kroku utworzony został magazynu zaufania w formacie PKCS12 przechowujący zasymulowany główny urząd certyfikacji, za pomocą którego podpisano certyfikat klienta.

Wdrożenie protokołu HTTPS do aplikacji opartej o technologię Spring Boot opiera się na umieszczeniu utworzonych plików `keystore.p12` (magazyn kluczy) oraz `truststore.p12` (magazyn zaufania) do katalogu zasobów oraz na odpowiednim skonfigurowaniu za pomocą pliku konfiguracyjnego, w który znajdują się takie informację jak, port na którym ma nasłuchiwać HTTPS, włączenie obsługi TLS, wymuszenie uwierzytelniania klienta, ścieżki do magazynu kluczy oraz zaufania, hasła do magazynów oraz typy magazynów. Zawartość pliku konfiguracyjnego dotyczącego kwestii związanych z protokołem HTTPS zamieszczono na Listing 1.

```
server:
  port: 443
  ssl:
    enabled: true
    client-auth: need
    key-store: src/main/resources/keystore/keystore.p12
    key-store-password: password
    key-store-type: PKCS12
    trust-store: src/main/resources/keystore/truststore.p12
    trust-store-password: password
    trust-store-type: PKCS12
```

Listing 1 Kod źródłowy sekcji pliku konfiguracyjnego dotyczącego protokołu HTTPS – opracowanie własne

8.1.2. Wdrożenie limitu zapytań

Wdrożenie zabezpieczeń dotyczących limitacji zapytań zostało zrealizowane za pomocą paradygmatu programowania zorientowanego na aspekty [4]. Jest to metodologia programowania, dzięki której możliwe jest wykonanie określonej czynności przed, po lub wokół konkretnej metody. Do zrealizowania zabezpieczenia w testowym interfejsie utworzone zostały dwa aspekty. Pierwszy z nich jest odpowiedzialny za ograniczenie ilości żądań do API pod punkt końcowy uwierzytelniania i odświeżania tokenu do pięciu na piętnaście minut. Drugi natomiast ma ograniczać ilość zapytań do pozostałych punktów końcowych do stu na minutę. Za pomocą Listing 2, zaprezentowany został kod źródłowy realizujący limitowanie zapytań z wykorzystaniem aspektów.

```
@Before("execution(*
pl.m4zek.springbootapi.api.controller.AuthController.*(..))")
public void limitRequestsAuthentication(JoinPoint joinPoint) throws
TooManyRequestsException {
    String key = joinPoint.getSignature().toLongString();
    checkLimitAndReset(key, RESET_INTERVAL_AUTHENTICATION,
LIMIT_PER_INTERVAL_AUTHENTICATION);
}

@Before("execution(* pl.m4zek.springbootapi.api.controller..*.*(..)) &&
!execution(*
pl.m4zek.springbootapi.api.controller.AuthController.*(..))")
public void limitOtherEndpoints(JoinPoint joinPoint) throws
TooManyRequestsException{
    String key = joinPoint.getSignature().toLongString();
    checkLimitAndReset(key,
       RESET_INTERVAL_OTHER_ENDPOINT, LIMIT_PER_INTERVAL_OTHER_ENDPOINT);
}

private void checkLimitAndReset(String key, Long RESET_INTERVAL, Integer
LIMIT_PER_INTERVAL) throws TooManyRequestsException {
    if (lastResetTime.containsKey(key) &&
         System.currentTimeMillis() - lastResetTime.get(key) >
                                                    RESET_INTERVAL) {
        requestCount.remove(key);
    }
    if (requestCount.containsKey(key)) {
        int count = requestCount.get(key);
        if (count >= LIMIT_PER_INTERVAL) {
            throw new TooManyRequestsException(
                              "Too many requests in a short time.
                               Pleas try in a while");
        } else {
            requestCount.put(key, count + 1);
        }
    } else {
        requestCount.put(key, 1);
    }
    lastResetTime.put(key, System.currentTimeMillis());
}
```

Listing 2 Kod źródłowy limitowania zapytań za pomocą aspektów – opracowanie własne

8.1.3. Wdrożenie walidacji danych wejściowych

W celu zrealizowanie walidacji danych wejściowych w Spring Boot, wykorzystane zostały adnotacje walidacji obiektów (ang. Bean Validation) [3], które mają za zadanie określać między innymi jaki format oraz jakie znaki ma posiadać zmienna w obiekcie, który jest mapowany na podstawie danych przesyłanych od klientów w formacie JSON. Przykładowe zastosowanie adnotacji zostało zamieszczone na Listing 3, który obrazuje jakie dane mogą zostać przesyłane do API pod punkt końcowy uwierzytelniania. Adnotacja `@NotBlank` oznacza, że pole nazwy użytkownika oraz hasło nie powinno być puste oraz nie może zawierać tylko znaków białych. Natomiast `@Pattern`, określa wzorzec jakie muszą posiadać dane wejściowe. W tym przypadku pole nazwy użytkownika i hasło może składać się jedynie z dowolnych znaków, które są literami włączając w to małe i duże litery oraz może zawierać cyfry. Dodatkowo przed obiektem, który mapowany jest w metodzie odpowiedzialnej za uwierzytelnianie dołączona jest adnotacja `@Valid`, określająca sprawdzenie obiektu za pomocą wcześniejszych adnotacji przed rozpoczęciem jej przetwarzania.

```
public class LoginModel {
    @NotBlank(message = "Username must not be empty")
    @Pattern(regexp = "^[a-zA-Z0-9]*$",
             message = "Username can only contains letters and numbers")
    private String username;

    @NotBlank(message = "Password must not be empty")
    @Pattern(regexp = "^[a-zA-Z0-9]*$",
             message = "Password can only contains letters and numbers")
    private String password;
}
```

Listing 3 Kod źródłowy walidujący dane dotyczące logowania użytkownika – opracowanie własne

Błędne dane wejściowe generują wyjątki, które należy obsłużyć i odesłać wiadomość zwrotną do klienta zawierająca treść spod atrybutu `message`. Do tego celu wykorzystano obsługę wyjątków oferowaną przez Spring Boot, przechwytującą błędy związane z walidacją. Listing 4 prezentuje kod źródłowy przechwytywania wyjątków związanych z walidacją danych.

```
@ExceptionHandler(ConstraintViolationException.class)
public ErrorMessage
handleArgumentNotValidException(ConstraintViolationException e,
WebRequest request){
    return new ErrorMessage(
            HttpStatus.BAD_REQUEST.value(), new Date(),
            e.getMessage().split(": ")[1], request.getDescription(true)
    );
}
```

Listing 4 Kod źródłowy realizujący przechwytywanie wyjątków związanych z walidacją danych – opracowanie własne

8.1.4. Wdrożenie szyfrowania tokenu JWT

Zrealizowanie zaszyfrowanego tokenu, wiąże się ze zmianą z podpisu cyfrowego na zaszyfrowanie tokenu. Do tego celu została zmieniona forma tokenu z JSON Web Token na JSON Web Encryption [18]. JWE jest pochodną JWT, która zamiast jedynie podpisywać dane cyfrowo oferuje ich szyfrowanie z wykorzystaniem różnych algorytmów. Dzięki wykorzystaniu JWE otrzymujemy zapewnienie poufności, integralności oraz autentyczności danych, które przechowywane są w formacie JSON. Nowy token bazuje na algorytmie symetrycznym AES-GCM, w którym ten sam klucz o długości dwustu pięćdziesięciu sześciu bitów służy do zaszyfrowania i deszyfrowania. Listing 5 przedstawia kod źródłowy realizujący utworzenie nowego zaszyfrowanego tokenu, który zawiera nazwę użytkownika.

```
public String generateToken(String username){

     return Jwts.builder()
             .subject(username)
             .issuedAt(new Date())
             .expiration(
                new Date(System.currentTimeMillis() + EXPIRATION * 1000))
             .encryptWith(key, alg, enc)
             .compact();
}
```

Listing 5 Kod źródłowy generowania nowego tokenu – opracowanie własne

8.1.5. Poprawienie konfiguracji mechanizmu CORS

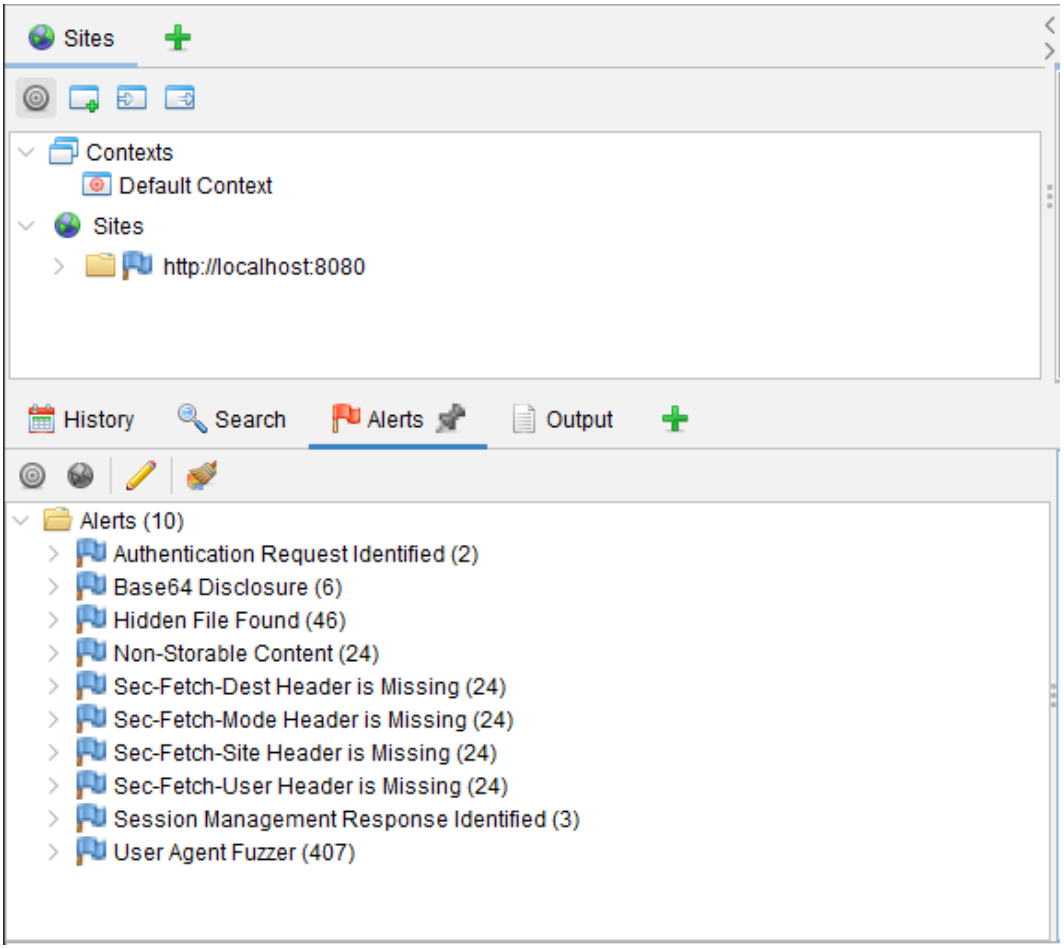
Poprawienie konfiguracji mechanizmu CORS jest związane z określeniem jakie domeny mogą mieć dostęp do interfejsu API. W tym celu wartość oznaczająca, że wszyscy mogą odwoływać się do API została zastąpiona adresem lokalnym, który przedstawiony jest na Listing 6.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
        @Override
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/api/**")
                    .allowedOrigins("http://127.0.0.1")
                    .allowedMethods(
                          "GET", "POST", "PUT", "DELETE", "PATCH")
                    .allowedHeaders(
                          "Origin", "X-Requested-With",
                          "Content-Type",  "Accept")
                    .allowCredentials(true)
                    .maxAge(3600);
        }
    };
}
```

Listing 6 Kod źródłowy poprawionej konfiguracji mechanizmu CORS – opracowanie własne

8.2. Ponowne aktywne skanowanie aplikacji pod kątem podatności

Aktywne skanowanie aplikacji zrealizowane zostało z takimi samymi ustawieniami jak w przypadku pierwszego skanowania, aby sprawdzić czy wdrożone udoskonalenia interfejsu API wyeliminowały wykryte podatności we wcześniejszym skanowaniu. Dodatkowo do ponownego skanowania konieczne było wyłączenie zabezpieczeń odnoszących się do limitowania zapytań oraz protokołu HTTPS ze względu na inne zachowania interfejsu po przekroczeniu limitu oraz braku możliwości nawiązania bezpiecznej komunikacji ze względu na dwustronny TLS, który wymaga uwierzytelnienia klienta.



Rys 20 Zrzut ekranu z aplikacji OWASP ZAP wyniki ponownego skanowania – opracowanie własne

Jak wynika z przedstawionego Rys 20, ponowne skanowanie nie wykryło nowych oraz wcześniejszych podatności. OWASP ZAP zwrócił jedynie alerty informujące, a nie alerty związane z brakami w zabezpieczeniach czy lukach w testowym interfejsie API. Oznacza to, że wdrożone zabezpieczenia spełniają swoją rolę.

9. Podsumowanie

Celem niniejszej pracy magisterskiej było zbadanie aspektów bezpieczeństwa testowego interfejsu API zrealizowanego w technologii Spring Boot. Praca opierała się na wykryciu potencjalnych podatności, które następnie zostaną zniwelowane poprzez wdrożenie mechanizmów bezpieczeństwa. Niniejszy ostatni rozdział pracy przedstawia podsumowanie osiągniętych rezultatów badawczych oraz zakończenie.

9.1. Podsumowanie osiągniętych rezultatów

W niniejszej pracy magisterskiej przeprowadzone zostały badania nad bezpieczeństwem testowego interfejsu API z wykorzystaniem testów. Dzieląc testy na dwa rodzaje z sukcesem został zrealizowany cel pracy magisterskiej polegający na wykryciu oraz eliminacji wykrytych podatności w testowym serwerze. Badania składały się z przeprowadzenia aktywnego skanowania oraz testów manualnych. Dzięki aktywnemu skanowaniu wykryte zostały takie podatności jak:

- Wstrzyknięcie złośliwego kodu SQL
- Błędna konfiguracja CORS
- Niezabezpieczone metody HTTP-DELETE

Dzięki sprawdzeniu powyższych luk za pomocą testów manualnych okazało się, że wstrzyknięcie złośliwego kodu SQL dało wynik fałszywie pozytywne dla wszystkich punktów końcowych, natomiast zostały wykryte ze względu na brak walidacji wejściowych, a to wiązało się z zapisaniem wstrzykniętego kodu SQL w bazie danych. Sprawdzenie podatności dotyczącej błędnej konfiguracji mechanizmu CORS dostarczyło wynik pozytywny, ze względu na ustawienie konfiguracji po stronie serwera zezwalającej na dostęp do zasobów ze wszystkich domen. Ostatnia wykryta podatność nawiązująca do niezabezpieczonej metody HTTP-DELETE, również okazała się wynikiem fałszywie pozytywnym. Dodatkowo przeprowadzono testy mechanizmu uwierzytelniania, dzięki którym wykryto dodatkowe podatności:

- Atak metodą siłową
- Brak zaszyfrowania danych w JWT
- Brak wdrożonego protokołu HTTPS

Wykorzystując wcześniej wykrytą podatność interfejsu API na ataki metodą siłową, zostały przeprowadzone testy wydajnościowe realizowane dla trzech scenariuszy – brak ataku, pojedynczy oraz potrójny atak metodą siłową. Wyniki testów ukazały, że pojedynczy atak

metody siłowej umożliwia w dalszym ciągu efektywne korzystanie z API, natomiast w przypadku scenariusza potrójnego znacznie traci na wydajności. Dodatkowo na bazie zebranych informacji zostały oszacowane potencjalne przepustowości interfejsu dla każdego badanego scenariusza.

Na podstawie wykrytych podatności zostały zrealizowane rekomendacje jakimi należy się kierować podczas tworzenia interfejsu API oraz dzięki nim wdrożone zostały mechanizmy bezpieczeństwa sukcesywnie niwelujące wykryte luki, które okazały się faktycznie istnieć w wyniku przeprowadzania testów manualnych. Dzięki wdrożeniu zabezpieczeń ponowne aktywne skanowanie interfejsu nie wykryło nowych podatności oraz tych, które zostały wykryte podczas pierwszego skanowania.

Kwestie związane z zabezpieczeniami interfejsów API, ale również wszelkich innych infrastruktur informatycznych, które są udostępniane w sieci są niezwykle ważne, ze względu na użytkowników z niej korzystających. Niniejsza praca magisterska skupiła się na przedstawieniu zarówno zagrożeń jak i konsekwencji, które mogą wynikać ze źle zabezpieczonych aplikacji. Ponadto przedstawia w jaki sposób i z jakich narzędzi można skorzystać w celu przetestowania różnych rodzajów aplikacji internetowych. Dzięki realizacji tej pracy dowiedziałem się jakie błędy popełniałem podczas realizacji aplikacji opartych na technologii Spring Boot oraz jak należy zabezpieczać interfejsy, aby posiadały wysoki stan zabezpieczeń. Należy jednak pamiętać, że testowanie bezpieczeństwa interfejsu API, ale również wszelkich aplikacji internetowych nie sprowadza się do przeprowadzania testów wyłącznie raz. Powinno się je wykonywać systematycznie, aby sprawdzać zabezpieczenia pod kątem starych oraz nowych potencjalnych podatności generujących zagrożenie.

Bibliografia

[1] Corey J. Ball: Hakowanie interfejsów API, Helion, 2023

[2] Yuri Diogenes, Erdal Ozkaya: Cyberbezpieczeństwo Strategie ataku i obrony – Jak osiągnąć najwyższy możliwy stan zabezpieczeń systemu informatycznego, Helion, 2023

[3] Craig Walls: Spring w akcji, Wydanie IV, Helion 2015

[4] Przemysław A. Bykowski: Spring boot LiveBook, LiveBooks, 2022

[5] Vijay Kumar Velu: Kali Linux i zaawansowane testy penetracyjne, Helion, 2023

[6] An incydent impacting some accounts and private information on Twitter https://privacy.twitter.com/en/blog/2022/an-issue-affecting-some-anonymous-accounts, *Źródło Internetowe, Data dostępu [06-12-2023]*

[7] T-Mobile hacked to steal data of 37 million accounts in API data breach, https://www.bleepingcomputer.com/news/security/t-mobile-hacked-to-steal-data-of-37-million-accounts-in-api-data-breach/*, Źródło Internetowe, Data dostępu [06-12-2023]*

[8] API attacks are on the rise, https://salt.security/api-security-trends*, Źródło Internetowe, Data dostępu [07-12-2023]*

[9] Burp Suite documentation: desktop editions, https://portswigger.net/burp/documentation/desktop*, Źródlo Internetowe, Data dostępu [22-12-2023]*

[10] OWASP ZAP Documentation, https://www.zaproxy.org/docs/*, Źródło Internetowe, Data dostępu [22-01-2023]*

[11] Postman documentation overview, https://learning.postman.com/docs/introduction/overview/*, Źródło Internetowe, Data dostępu [23-12-2023]*

[12] VirtualBox: Oracle VM VirtualBox User Manual, https://www.virtualbox.org/manual/UserManual.html*, Źródło Internetowe, Data dostępu [03-01-2024]*

[13] MariaDB Server (SQL Database Server), https://mariadb.com/docs/server/*, Źródło Internetowe, Data dostępu [03-01-2024]*

[14] Kiterunner, https://github.com/assetnote/kiterunner*, Źródło Internetowe, Data dostępu [22-01-2024]*

[15] OWASP Amass User Guide, https://github.com/owasp-amass/amass/blob/master/doc/user_guide.md*, Źródło Internetowe, Data dostępu [22-01-2024]*

[16] Rolf Oppliger: SSL and TLS Theory and Practice, Artech House 2023

[17] Cay S. Hortsman: Java Techniki Zaawansowane, Wydanie XI, Helion, 2019

[18] Java JWT: JSON Web Token for Java and Android, https://github.com/jwtk/jjwt, *Źródło Internetowe, Data dostępu [22-01-2024]*

Spis Rysunków

Spis Tabel

Spis Listingów