JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 252–260
Received: 20 May 2020
Accepted: 28 May 2020

A security analysis of authentication and authorization implemented in web applications based on the REST architecture

Analiza bezpieczeństwa mechanizmów uwierzytelniania oraz autoryzacji implementowanych w aplikacjach internetowych zbudowanych w oparciu o architekturę REST

Tomasz Muszyński*, Grzegorz Kozieł

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The purpose of this article is to prepare a security analysis of authentication and authorization mechanisms in web applications based on the REST architecture. The article analyzes the problems encountered during the implementation of the JSON Web Token (JWT) mechanism. The article presents examples of problems related to the implementation of authorization and authentication, and presents good practices that help ensure application security.
Keywords: security; security vulnerability; REST; JWT

Streszczenie

Celem artykułu jest analiza bezpieczeństwa mechanizmów uwierzytelniania oraz autoryzacji w aplikacjach internetowych zbudowanych w oparciu o architekturę REST. W artykule przeanalizowano problemy spotykane podczas implementacji mechanizmu JSON Web Token (JWT). W artykule podano przykłady problemów związanych z wdrożeniem autoryzacji i uwierzytelniania oraz przedstawiono dobre praktyki ułatwiające zapewnienie bezpieczeństwa aplikacji.
Słowa kluczowe: bezpieczeństwo; podatność bezpieczeństwa; REST; JWT

*Corresponding author

Email address: **tomasz.muszynski@pollub.edu.pl** (T. Muszyński)

©Published under Creative Common License (CC BY-SA v4.0)

1. Wstęp

Współcześnie trudno spotkać oprogramowanie, które w swojej architekturze nie realizuje procesu uwierzytelnienia oraz autoryzacji.

Uwierzytelnianie to proces, którego celem jest weryfikacja czy zebrane od podmiotu poświadczenia potwierdzają jego tożsamość. Natomiast proces autoryzacji na podstawie zapisanych w systemie uprawnień podejmuje decyzję o odmowie lub przyznaniu dostępu do zasobów. Proces autoryzacji bazuje na zweryfikowanej tożsamości użytkownika.

Niepoprawna implementacja mechanizmów uwierzytelnienia oraz autoryzacji jest częstym powodem problemów bezpieczeństwa w przypadku aplikacji zbudowanych w architekturze REST [1]. Realizacja powyższych procesów najczęściej wykorzystuje mechanizm JSON Web Token, którego złożoność może być przyczyną popełnianych błędów a w konsekwencji wystąpienia podatności bezpieczeństwa. W artykule omówiono kontekst bezpieczeństwa JWT oraz problemy związane z implementacją uwierzytelniania oraz autoryzacji w aplikacjach zbudowanych w architekturze REST.

2. Cel publikacji

Celem publikacji jest przeprowadzenie analizy bezpieczeństwa procesów uwierzytelniania oraz autoryzacji implementowanych w aplikacjach internetowych zbudowanych w architekturze REST.

3. Przegląd literatury

Bezpieczeństwo aplikacji zbudowanych w architekturze REST API w niewielu publikacjach jest traktowane jako oddzielna kategoria poddana analizie. W literaturze polskojęzycznej trudno znaleźć opracowania tej tematyki, dopiero w 2019 roku ukazała się książka pt. „Bezpieczeństwo aplikacji webowych” wydana przez wydawnictwo SECURITUM, w której znajduje się rozdział poświęcony bezpieczeństwu architektury REST. W rozdziale pt. „Bezpieczeństwo API REST”, autor zauważa podobieństwa aplikacji w architekturze REST do pozostałych aplikacji internetowych, natomiast wskazuje również różnice, które powodują, że warto ten temat analizować jako oddzielną kategorię. W rozdziale można znaleźć informację o przyczynach błędów bezpieczeństwa takich jak:

- brak dokładnego określenia jakie metody powinny być użyte do jakich operacji,
- użycie frameworków lub bibliotek zawierających podatności,
- włączony tryb „Debug”,
- różnorodność formatów danych,
- niepoprawna implementacja uwierzytelnienia i autoryzacji, częsty brak wdrożonego uwierzytelnienia.

Autor rozdziału wspomina również o małej liczbie publikacji badających bezpieczeństwo tej architektury. „Osobiście uważam, że zagadnienie jest jeszcze mało zbadane, niewiele mamy również dostępnych informacji

o konkretnych podatnościach w API REST – np. znalezionych w ramach programów bug bounty.”[1].

Kolejnym rozdziałem we wspomnianej książce jest rozdział opisujący niebezpieczeństwo związane z wdrożeniem JSON Web Token (JWT), użycie JWT jest powszechne w procesie wymiany danych uwierzytelniających implementowanych w REST API. W rozdziale można znaleźć analizę aż jedenastu problemów bezpieczeństwa JWT oraz praktyczne rady jak im przeciwdziałać. Temat bezpieczeństwa JWT to podstawowa kwestia związana z problemami spotykanymi w aplikacjach opartych na architekturze REST.

Autorzy badający bezpieczeństwo REST API w dużej mierze skupiają się na problemach, które związane są z implementacją JWT. W artykule pt. „Microservices API Security”[2] autor wspomina o braku szyfrowania JWT. JWT jest podpisane, jednak nie są automatycznie szyfrowane (szyfrowanie JWT jest funkcją opcjonalną). To znaczy że wszelkie dane znajdujące się w tokenie mogą być odczytane przez każdego, kto uzyska do niego dostęp. Autor również jako problem opisuje brak możliwości natychmiastowego zablokowania użytkownika poprzez anulowania JWT, ponieważ token przechowywany jest po stronie klienta.

Autor artykułu „The Protection of Information in Computer Systems”[3] wskazuje na to, że REST API powinno być bezstanowe, w każdym żądaniu powinien być przesyłany komplet informacji pozwalający na ponowne weryfikowanie czy przesłane informacje umożliwiają dostęp do systemu, co również wiąże się z wybraniem JWT jako mechanizmu pozwalającego na dołączanie danych uwierzytelniających w każdym zapytaniu. Autor omawia dobre praktyki projektowania zabezpieczeń, które warto wdrożyć w REST API, opisywane zasady wskazują miedzy innymi, że:

- użytkownik powinien mieć tylko takie uprawnienia, które są niezbędne do wykonywania przez niego zadań, uprawnienia powinny być rozszerzane wyłącznie w ramach potrzeb,
- nie powinno stosować się domyślnych poziomów uprawnień,
- system powinien być możliwie prosty, dzięki temu będzie zawierał mniej błędów,
- każde zapytanie powinno być weryfikowane pod względem uprawnień bazując na rzeczywistym stanie, a nie zbuforowanych danych, tak aby odwołanie uprawnień działało natychmiast.

Kolejnym źródłem wiedzy o bezpieczeństwie aplikacji internetowych są publikacje organizacji OWASP (ang. Open Web Application Security Project). OWASP wydaje specjalne dokumenty miedzy innymi zawierające opis najbardziej krytycznych podatności aplikacji internetowych, organizacja postanowiła wydzielić kategorie podatności API jako oddzielny zbiór. Opisane w publikacji podatności można uznać za ściśle związane z implementacja API w standardzie REST. Wydzielenie tematyki API przez organizację OWASP udowadnia, że ten temat zasługuje na oddzielną analizę.

OWASP opracował dwa dokumenty: „Application Security Verification Standard 4.0”[4] w skrócie określany ASVS oraz „Top 10 Proactive Controls”[5], dokumenty te przeznaczone są jako pomoc w implementacji i w testowaniu bezpieczeństwa aplikacji internetowych. W powyższych dokumentach można znaleźć informacje na temat wymagań, które należy zweryfikować podczas pracy z oprogramowaniem zbudowanym w architekturze REST. Publikacja zawiera wskazania na potrzebę następujących weryfikacji:

- poprawność wprowadzanych danych, wdrożenie odpowiedniej walidacji, sprawdzanie typu danych wejściowych,
- czy komunikacja jest szyfrowana,
- czy JWT zostało prawidłowo zaimplementowane.

W wyniku analizy bieżącego stanu wiedzy należy stwierdzić, że wydzielenie aplikacji zbudowanych w architekturze REST z pośród wszystkich aplikacji internetowych jest zasadne.

Często omawianym słabym punktem związanym z aplikacjami wykorzystującymi REST jest zastosowanie złożonego mechanizmu JWT. Pomimo, że wiele występujących podatności we współczesnych aplikacjach internetowych spowodowanych jest takimi czynnikami jak przeoczenie, niedbałość, brak testowania, to wybór architektury REST może mieć konsekwencje, o których warto wiedzieć podczas projektowania i implementacji.

4. REST

Architektura oprogramowania określa organizację systemu i jego komponentów, definiuje reguły budowy oraz zasady rozwoju. REST (ang. REpresentational State Transfer) jest stylem architektury oprogramowania dedykowanym dla rozproszonych systemów, został zaprezentowany przez Roya Fieldinga w 2000 roku [6]. Z architekturą REST ściśle wiąże się pojęcie API (ang. Application Programming Interface). API jest zbiorem reguł określających zasady komunikacji pomiędzy programami komputerowymi. REST API w ostatnich latach zyskało dużą popularność, wypierając alternatywną metodę komunikacji jaką jest wykorzystanie protokołu SOAP. Rysunek 1 przedstawia zainteresowanie architekturą REST w porównaniu z protokołem SOAP w ujęciu czasowym według wyszukiwarki Google.

Rysunek 1: Porównanie zainteresowania REST API z SOAP [7]

Autor REST zdefiniował sześć następujących ograniczeń:

- bezstanowość (ang. stateless) – każde żądanie jest niezależne od poprzedniego, jest to też własność protokołu HTTP,
- jednolity interfejs (ang. Uniform interface) – jednolity interfejs definiowany jest przez cztery założenia.

Po pierwsze adresy powinny jednoznacznie wskazywać do jakiego zasobu się odwołują. Drugie ograniczenie mówi o tym, że zasoby mogą być modyfikowane przez klienta, posiadającego reprezentację zasobu i dołączone do niego metadane. Trzecie ograniczenie wskazuje, że wiadomości wymieniane powinny zawierać taką ilość informacji, która pozwoli serwerowi na dokładne ich przetworzenie. Ostatnie ograniczenie określa, że serwer może odpowiadać tekstem zawierającym hiperłącza,

- pamięć podręczna (ang. cache) – serwer w odpowiedzi powinien określić czy odpowiedź może zostać buforowana. Dane zwracane przez interfejs mogą być danymi, które często ulegają zmianie wtedy takie dane warto oznaczyć jako niebuforowane. Istnieją też dane, które nie zmieniają się lub zmieniają się rzadko w takich przypadkach zastosowanie buforowania może w znacznym stopniu zmniejszyć liczbę połączeń z serwerem, przez co pozwoli zminimalizować wykorzystywane zasoby,
- klient-serwer (ang. client-server) – odseparowanie zadań wykonywanych przez serwer od interfejsu użytkownika, zwiększa możliwości skalowalności i ułatwia dostarczanie rozwiązań na różne platformy,
- system warstwowy (ang. layered system) – architektura składa się z wydzielonych warstw, każda z warstw ma oddzielne zadania i prowadzi interakcje wyłącznie z przyległymi warstwami,
- kod na żądanie (ang. code on demand) – jest opcjonalnym ograniczeniem, REST pozwala klientowi na rozszerzanie funkcjonalności poprzez odbieranie i wykonywanie kodu w postaci skryptów lub apletów zwiększając funkcjonalność systemu.

Wszystkie informacje związane z REST określane są jako zasób. Zasobem może być tekst, plik multimedialny, użytkownik. Stan zasobu jest określany jako reprezentacja zasobu. Reprezentacja zbudowana jest z danych, hiperłączy, oraz metadanych ich opisujących [6].

5. JSON Web Token

JSON Web Token w skrócie JWT, jest otwartym standardem zdefiniowanym w dokumencie RFC 7519 [8]. JWT został opracowany w celu bezpiecznego przekazywania danych pomiędzy komunikującymi się usługami. Bezpieczeństwo może zostać zrealizowane na dwóch płaszczyznach, pierwsza to integralność danych. Integralność realizowana jest przy wykorzystaniu podpisów. Drugim aspektem bezpieczeństwa jest poufność, dane przekazywane przy użyciu JWT mogą zostać zaszyfrowane [9]. Dane JWT mogą zostać zakodowane w jedną z dwóch struktur:

- JWS (ang. JSON Web Signature),
- JWE (ang. JSON Web Encryption).

W praktyce często JWS nazywany jest jako JWT [1].

JWT ze względu na swoją specyfikę jest najczęściej wykorzystywany w dwóch procesach implementowanych w oprogramowaniu:

- autoryzacja – po udanym uwierzytelnieniu, serwer przesyła do klienta JWT, który może zostać użyty w kolejnych żądaniach jako potwierdzenie tożsamości uprawniające do operacji na autoryzowanych zasobach,
- wymiana informacji – dane są przesyłane między stronami przy wykorzystaniu par kluczy publiczny - prywatny, zapewnia to bezpieczne przesyłanie informacji [9].

JWS składa się z trzech oddzielonych kropkami elementów, każdy z elementów zakodowany jest algorytmem Base64URL:

- nagłówek – zbudowany jest z dwóch elementów określających typ tokena oraz algorytm podpisu,
- payload – zawiera docelowe dane, które będą istotne z punktu widzenia realizowanej funkcjonalności, w ładunku przesyłane są zazwyczaj dane identyfikujące użytkownika systemu, oraz dane pomocnicze takie jak czas ważności,
- podpis – gwarantuje integralność danych, na podpis składa się nagłówek, payload oraz tajny klucz przechowywany po stronie serwera. Podpis jest wynikiem działania algorytmu określonego w nagłówku w parametrze „alg”[9].

Przykładowy JWT został zaprezentowany na Rysunku 2.

Rysunek 2: Przykład JWT [1]

5.1. Kontekst bezpieczeństwa

Struktura JWT jest skomplikowana i nie zawsze jest dobrze zrozumiana przez osoby implementujące jej wykorzystanie. Dane przesyłane w JWS nie są szyfrowane i mogą zostać odczytane , dlatego też w JWS nie powinny znajdować się poufne informacje, które należy chronić, dopiero struktura JWE implementuje mechanizmy szyfrowania danych i jest w stanie zagwarantować ich poufność. JWS umożliwia zapewnienie integralności danych, która realizowana jest przez podpis Jeżeli atakujący zmieni payload podpis nie zostanie poprawnie zweryfikowany przez serwer i JWT zostanie odrzucony. Zmiana podpisu przez atakującego nie jest możliwa ponieważ elementem podpisu jest tajny klucz znajdujący się po stronie serwera. JWT dostarcza dużą elastyczność przy zapewnieniu bezpieczeństwa, natomiast ze względu na złożoność implementacji mogą wystąpić błędy, w wyniku których pojawią się podatności bezpieczeństwa.

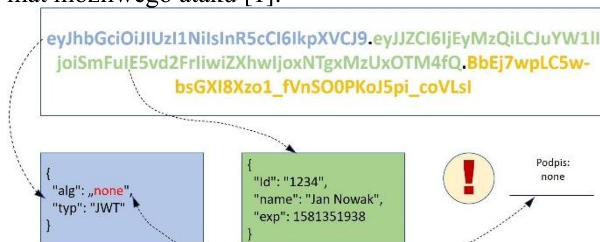
Dokumentacja opisująca specyfikę JWT wskazuje na możliwość użycia JWT bez podpisu, brak podpisu określany jest w nagłówku, który może wyglądać jak na Listingu 1.

Listing 1: Nagłówek JWT określający brak wymagania podpisu

```
{
  "alg": "none",
  "typ": "JWT"
}
```

Atakujący może wykorzystać ten fakt i spróbować zmienić algorytm podpisu na „none", jeżeli zaimplementowana weryfikacja JWT nie uwzględnia takiej sytuacji, napastnikowi może udać się dostać do nieprzeznaczonych dla niego danych.

Podatność występuje w bibliotekach implementujących JWT, biblioteki niepoprawnie realizują funkcjonalność weryfikacji podpisu ponieważ nie są odporne na zmianę typu algorytmu. Atakujący zmienia zawartość payloadu a następnie modyfikuje algorytm podpisu, w kolejnym kroku JWT wysyłany jest w żądaniu a serwer podejmuje próbę weryfikacji. Serwer powinien zwrócić błąd weryfikacji, natomiast ze względu na luki nie zawsze tak się dzieje. Rysunek 3 przedstawia schemat możliwego ataku [1].



Rysunek 3: Schemat ataku [1]

W celu zobrazowania przebiegu ataku wykorzystano środowisko NodeJS w wersji 10.16.3, oraz bibliotekę „jsonwebtoken" w wersji 8.5.1.

Istotę podatności można przedstawić za pomocą prostego kodu programu. W pierwszym kroku tworzony jest nowy JWT, następnie wykonywane jest jego dekodowanie oraz weryfikacja. Kolejnym etapem jest zmiana na algorytmu na algorytm „none", następnie ponownie wykonywane jest dekodowanie i weryfikacja. Przykład obrazujący na czym polega luka zaprezentowano na Listingu 2.

Listing 2: Kod JavaScript obrazujący podatność

```
const jwt = require('jsonwebtoken');
const base64url = require('base64url');

const privateKey = '123456789abcdefgh';

//Utworzenie podpisanego tokenu
let token = jwt.sign({ id: '1234', name: 'Jan Nowak', admin: false },
privateKey, { algorithm: 'HS256'});

// Przed ingerencją w token
const decodedToken = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken);

const verifiedToken = jwt.verify(token, privateKey);
console.log('Zweryfikowany token: ');
console.log(verifiedToken);

//Zmiana algorytmu
let tokenHeader = token.split('.')[0];
tokenHeader = JSON.parse(base64url.decode(tokenHeader));
console.log('Nagłówek przed zmianą algorytmu: ');
console.log(tokenHeader);
console.log('Nagłówek po zmianie algorytmu: ');
tokenHeader.alg = 'none';
console.log(tokenHeader);
const newHeader =base64url(JSON.stringify(tokenHeader));
token = newHeader + "." + token.split('.')[1] + "." +
token.split('.')[2];
// Po ingerencji w token
const decodedToken2 = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken2);

console.log('Zweryfikowany token: ');
try {
  const verifiedToken2 = jwt.verify(token, privateKey);
  console.log(verifiedToken2);
} catch(err) {
  console.log('Błąd weryfikacji: ' + err.message);
}
```

Wykorzystana w przykładzie wersja biblioteki okazała się odporna na opisywany atak, natomiast w innych bibliotekach lub własnych implementacjach taki atak może zakończyć się powodzeniem. Wynik działania programu przedstawiono na Rysunku 4.

```
$ node app.js
Zawartość tokenu:
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587296819 }
Zweryfikowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587296819 }
Nagłówek przed zmianą algorytmu:
{ typ: 'JWT', alg: 'HS256' }
Nagłówek po zmianie algorytmu:
{ typ: 'JWT', alg: 'none' }
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587296819 }
Zweryfikowany token:
Błąd weryfikacji: invalid signature
```

Rysunek 4: Zrzut z konsoli prezentującej wynik działania programu

Użyta w przykładzie biblioteka „jsonwebtoken" jest odporna na zmianę rodzaju algorytmu podpisu, nie oznacza to, że inne biblioteki operujące na JWT będą równie dobrze zabezpieczone przed tym rodzajem błędów. Skutkiem ignorowania podpisu może być całkowite przejęcie konta użytkownika. Napastnik może dowolnie zmienić ładunek JWT bez wykrycia tego przez mechanizmy weryfikujące.

Kolejnym problemem może być błędna implementacja procesu weryfikacji, która polega na zastosowaniu nieodpowiedniej metody. JWT jest tylko dekodowany bez weryfikowania czy nie został zmodyfikowany przez użytkownika. W celu zobrazowania podatności wykorzystano środowisko NodeJS w wersji 10.16.3, oraz bibliotekę „jsonwebtoken" w wersji 8.5.1. Poniżej przedstawiono kroki, których wykonanie umożliwia zobrazowanie i zrozumienie przyczyn występowania podatności. Pierwszym krokiem jest utworzenie projektu, w tym celu wykonujemy polecenie npm init -y, wynik polecenia zobrazowano na Rysunku 5.

```
$ npm init -y
wrote to J:\aplikacja\vulnerability_1\package.json:

{
  "name": "vulnerability_1",
  "version": "1.0.0",
  "descriprion": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Rysunek 5: Utworzenie projektu w NodeJS

Następnie należy zainstalować bibliotekę wspomagającą implementację JWT. Proces instalacji biblioteki zobrazowano na Rysunku 6.

```
$ npm install jsonwebtoken
+ jsonwebtoken@8.5.1
added 15 packages from 10 contributors and audited 17 packages in 3.773s
found 0 vulnerabilities
```

Rysunek 6: Instalacja biblioteki „jsonwebtoken

W celu zmiany zawartości payloadu skorzystano z biblioteki „base64url". Rysunek 7 zawiera szczegóły instalacji.

```
$ npm i base64url
+ base64url@3.0.1
added 1 packages from 1 contributors and audited 18 packages in 4.61s
found 0 vulnerabilities
```

Rysunek 7: Instalacja biblioteki base64url

Kod programu znajdujący się na Listingu 3 tworzy nowy JWT, wykonuje jego dekodowanie i weryfikację, następnie zawartość payloadu jest zmieniana i ponownie wykonywany jest proces dekodowania i weryfikacji. Użycie dekodowania, które nie weryfikuje podpisu a jedynie odczytuje zawartość zamiast weryfikacji umożliwia atakującemu zmianę danych przesyłanych w tokenie.

Listing 3: Kod JavaScript przedstawiający różnicę pomiędzy weryfikacją a dekodowaniem

```
const jwt = require('jsonwebtoken');
const base64url = require('base64url');

const privateKey = '123456789abcdefgh';

//Utworzenie podpisanego tokenu
let token = jwt.sign({ id: '1234', name: 'Jan Nowak', admin: false },
privateKey, { algorithm: 'HS256'});

// Przed ingerencją w token
const decodedToken = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken);

const verifiedToken = jwt.verify(token, privateKey);
console.log('Zweryfikowany token: ');
console.log(verifiedToken);

//Zmiana payloadu
let tokenPayload = token.split('.')[1];
tokenPayload = JSON.parse(base64url.decode(tokenPayload));
tokenPayload.admin = true;

const newPayload =base64url(JSON.stringify(tokenPayload));
token = token.split('.')[0]+ "." + newPayload + "." +
token.split('.')[2];

// Po ingerencji w token
const decodedToken2 = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken2);

console.log('Zweryfikowany token: ');
try {
  const verifiedToken2 = jwt.verify(token, privateKey);
  console.log(verifiedToken2);
} catch(err) {
  console.log('Błąd weryfikacji: ' + err.message);
}
```

Dekodowanie nie weryfikuje podpisu, dopiero proces weryfikacji zwraca błąd i uniemożliwia eskalację uprawnień. Wynik działania programu został przedstawiony na Rysunku 8.

```
$ node app.js
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587293399 }
Zweryfikowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587293399 }
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: true, iat: 1587293399 }
Zweryfikowany token:
Błąd weryfikacji: invalid signature
```

Rysunek 8: Wynik porównania weryfikacji i dekodowania

Zastosowanie dekodowania zamiast weryfikacji może mieć poważne skutki, atakujący może zmienić dane dostarczane w tokenie i zwiększyć uprawnienia lub podszyć się pod innego użytkownika systemu. Wystąpienie podatności może być wynikiem braku znajomości zewnętrznych bibliotek. Dokumentacja wyraźnie opisuje skutki wybrania funkcji dekodowania zamiast weryfikacji, Rysunek 9 przedstawia fragment dokumentacji biblioteki „jsonwebtoken".

jwt.decode(token [, options])

(Synchronous) Returns the decoded payload without verifying if the signature is valid.

> **Warning:** This will **not** verify whether the signature is valid. You should **not** use this for untrusted messages. You most likely want to use `jwt.verify` instead.

Rysunek 9: Fragment dokumentacji biblioteki „jsonwebtoken" [10]

Jeżeli metoda weryfikacji zostanie poprawnie wybrana oraz zastosuje się odpowiednie zabezpieczenia przed zmianą algorytmu na „none" można napotkać na kolejną trudność polegającą na odpowiednim doborze klucza. Podpis JWT składa się z nagłówka, paylodu oraz tajnego klucza przechowywanego po stronie serwera, wybranie klucza o krótkiej długości może spowodować szybkie jego złamanie. Wszystkie dane potrzebne do próby złamania tajnego klucza znajdują się po stronie użytkownika, czyli osoby atakującej. Wykrycie próby forsowania klucza nie jest możliwe przez serwer ponieważ wszystkie operacje wykonywane są poza serwerem. Przykładowym narzędziem umożliwiającym łamanie tajnego klucza JWT jest narzędzie JWT cracker [11].

W celu zademonstrowania problemu przygotowano JWT ze słabym kluczem tajnym o wartości 1234. W celu przygotowania JWT użyto narzędzia dostępnego pod adresem https://jwt.io/, proces tworzenia JWT zaprezentowano na Rysunku 10.

Rysunek 10: Przygotowanie JWT ze słabym kluczek tajnym

Skrypt umożliwiający złamanie klucza, wykorzystuje narzędzie JWT cracker oraz wyświetla czas rozpoczęcia i czas zakończenia procesu. Treść skryptu została przedstawiona na Listingu 4.

Listing 4 Skrypt łamiący klucz tajny JWT

```
#!/bin/sh
# Skrypt łamiący klucz tajny JWT
date +"%T"
./jwtcrack eyJhbGciOiJIUzI1NiI
sInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3
ODkwIiwibmFtZSI6IkphbiBOb3dhayIsImlhdCI6MTUxNjIzO
TAyMn
0.JuFvEJoJk9BoBZD3vT5NCy5y4SOALp9Pp8mwpvLO3k
date +"%T"
```

Skrypt po czasie nie dłuższym niż dwie minut zwrócił tajny klucz. Wynik działania programu przedstawiono na Rysunku 11.

Rysunek 11: Wynik skryptu łamiącego klucz tajny

Fakt, że dzięki JWT możemy zrealizować uwierzytelnienie bez komunikacji z bazą danych to podstawa kolejnego problemu bezpieczeństwa. Jeżeli tokeny nie będą miały określonych krótkich dat ważności oraz w mechanizmie weryfikującym ważność tokenu nie będzie właściwie sprawdzana, nie będzie możliwa natychmiastowa zmiana uprawnień lub zablokowanie użytkownika. Częstym rozwiązaniem tego problemu jest generowanie dwóch tokenów jeden, który zawiera dane o użytkowniku (token uwierzytelniający) z krótkim czasem ważności i drugi, tak zwany token odświeżającym, który zapisywany jest w bazie danych. Token odświeżający służy do pozyskania nowego tokenu autoryzującego w chwili jego unieważnienia.

6. Autoryzacja oraz uwierzytelnianie

Opisane poniżej podatności związane są z niepoprawnym wdrożeniem mechanizmów uwierzytelniania oraz autoryzacji.

Szczegóły podatności zostały zademonstrowane w oparciu o aplikację REST API. Aplikacja została opracowaną w technologii NodeJS przy wykorzystaniu frameworka Express.

6.1. Autoryzacja na poziomie obiektu

Autoryzacja na poziomie obiektu dotyczy sytuacji, w której użytkownik jest uwierzytelniony i próbuje uzyskać dostęp do konkretnego obiektu. Decyzja o przyznaniu lub odmowie dostępu podejmowana jest na podstawie zapisanych w bazie danych poświadczeń. Atakujący próbuje modyfikować parametry zapytania np. identyfikator obiektu w ten sposób aby w odpowiedzi zostały zwrócone dane, które nie powinny być mu zaprezentowane.

Napastnik zarejestrował się oraz uwierzytelnił, następnie przeprowadza rekonesans, w wyniku którego znalazł punkt końcowy odpowiedzialny za pobranie informacji na temat jego profilu. Żądanie pobierające dane użytkownika przedstawia Listing 5.

Listing 5 Żądanie pobrania informacji o użytkowniku

```
GET /api/users/v1/1 HTTP/1.1
Host: localhost:3010
Accept: application/json, text/plain, */*
Content-Type: application/json
```

W odpowiedzi na powyższe żądanie serwer zwraca informacje na temat użytkownika, odpowiedź serwera przedstawia Listing 6.

Listing 6 Odpowiedź zawierająca dane użytkownika

```
HTTP/1.1 200 OK
[Nagłówki]

{
  "id":1,
  "email": "jan@niepodam.pl",
  "first_name": "Jan",
  "last_name": "Kowalski",
  "id_number": "ABC12345",
  "pesel": "12345612345",
  "phone_number": "111 222 333",
  "password":
"$2a$10$.Ci9NPsLJrppf.81MOoa.uXKghsnsMQcQpc62Zk53szCsnp
Yc1Xf2",
  "active": false,
  "created_at": "2020-05-16T09:31:00.069Z"
}
```

Identyfikator przesyłany w żądaniu ma wartość 1, jest to identyfikator użytkownika, podając inną wartość parametru proces autoryzacji powinien zwrócić odpowiedź z informacją o braku uprawnień. Ponieważ autoryzacja nie została odpowiednio zaimplementowana podanie identyfikatora należącego do innego użytkownika spowoduje przetworzenie zapytania i zwrócenie odpowiedzi z pełnymi danymi. Rysunek 12 przedstawia zmodyfikowane zapytanie wysłane przy pomocy programu Burp Suite.

Rysunek 12 Zmodyfikowane żądanie

W odpowiedzi na żądanie została zwrócona odpowiedź zawierająca dane innego użytkownika (Rysunek 13), dzieje się tak, ponieważ serwer polega na pa-

rametrach, które przesyła użytkownik, a nie na jego uprawnieniach.

```
1 HTTP/1.1 200 OK
2 X-DNS-Prefetch-Control: off
3 X-Frame-Options: SAMEORIGIN
4 Strict-Transport-Security: max-age=15552000; includeSubDomains
5 X-Download-Options: noopen
6 X-Content-Type-Options: nosniff
7 X-XSS-Protection: 1; mode=block
8 Content-Type: application/json; charset=utf-8
9 Content-Length: 603
10 ETag: W/"25b-oTcm2C63x+a6FYj3FmIaD2EM1Hw"
11 Date: Thu, 21 May 2020 14:26:00 GMT
12 Connection: keep-alive
13
14 {
     "response":{
       "id":2,
       "email":"michal@niepodam.pl",
       "first_name":"Michał",
       "last_name":"Nowak",
       "id_number":"CBD 23456",
       "pesel":"77777766666",
       "phone_number":"222 333 444",
       "password":"$2a$10$2jE5LtLpvCICjQvWx/04X.9IjHykypDuo6bNZDNUpOKkRBGHGNd8C",
       "active":false,
       "created_at":"2020-05-16T09:34:43.799Z",
       "isAdmin":false
     },
     "sql":"{ method: 'select',\n        options: {},\n        timeout: false,\n
```

Rysunek 13 Odpowiedź na zmodyfikowane żądanie

Mechanizmy kontrolujące dostęp użytkowników do obiektów zazwyczaj są mechanizmami złożonymi, przez co są bardziej podatne na luki bezpieczeństwa. Należy bezwzględnie zweryfikować, czy mechanizm autoryzacji bazuje na uprawnieniach użytkowników.

6.2. Błędy w uwierzytelnianiu użytkowników

Punkty końcowe odpowiedzialne za proces uwierzytelniania są dostępne na zewnątrz, przez co często są pierwszym wektorem ataku. W przypadku przełamania uwierzytelniania atakujący może przejąć konto, które zawiera poufne dane oraz uprawnia do wykonywania operacji modyfikujących lub trwale usuwających dane.

Przykładowym problemem, który można zaliczyć do tej kategorii jest brak odpowiedniej polityki bezpieczeństwa haseł, Podczas rejestracji akceptowalne są krótkie hasła o małej złożoności w wyniku czego możliwe jest przeprowadzenie skutecznego ataku siłowego. Listing 7 zawiera treść zapytania, umożliwiającego uwierzytelnienie użytkownika.

Listing 7 Treść zapytania wykorzystywanego do uwierzytelnienie

```
POST /api/users/login HTTP/1.1
Host: localhost:3010
Accept: application/json, text/plain, */*
Content-Type: application/json

{
    "email": "jan@niepodam.pl",
    "password": "*******"
}
```

W ciele żądania przekazywany jest adres e-mail oraz hasło użytkownika, po odebraniu żądania przez serwer następuje weryfikacja i podejmowana jest decyzja o uwierzytelnieniu. Uwierzytelnienie polega na zwróceniu JWT, który będzie dołączany do kolejnych żądań w celu potwierdzenia tożsamości.

Do przeprowadzenia ataku siłowego na hasło można wykorzystać narzędzie Intruder z pakietu Burp Suite. Rysunek 14 przedstawia odpowiednio przygotowaną treść zapytania.

Payload Positions

Configure the positions where payloads will be inserted into the base request. The a

Attack type: Sniper

```
1 POST /api/users/login HTTP/1.1
2 Host: localhost:3010
3 Accept: application/json, ext/plain, */*
4 Content-Type: application/json
5
6 {
7   "email": "marek@niepodam.pl",
8   "password": "§§"
9 }
```

Rysunek 14 Treść sparametryzowanego żądania uwierzytelnienia

W miejscu hasła zostaną wstawione wartości zdefiniowane w opcjach przedstawionych na Rysunku 15.

Payload Options [Brute forcer]

This payload type generates payloads of specified lengths that contain all permutations

Character set: abcdefghijklmnopqrstuvwxyz0123456789

Min length: 4

Max length: 4

Rysunek 15 Definicja wartości parametrów wstawianych w żądaniu

W wyniku przeprowadzonego ataku siłowego zostało odgadnięte hasło, które umożliwi przejęcie konta użytkownika i wykonywanie operacji w jego imieniu. Rysunek 16 prezentuje wynik ataku. Hasło użytkownika ma wartość „ndaa".

Results | Target | Positions | Payloads | Options

Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length	Comment
122	ndaa	200	☐	☐	859	
0		403	☐	☐	470	
1	aaaa	403	☐	☐	470	
2	baaa	403	☐	☐	470	

Request | Response

Raw | Headers | Hex | Render

```
1 HTTP/1.1 200 OK
2 X-DNS-Prefetch-Control: off
3 X-Frame-Options: SAMEORIGIN
4 Strict-Transport-Security: max-age=15552000; includeSubDomains
5 X-Download-Options: noopen
6 X-Content-Type-Options: nosniff
7 X-XSS-Protection: 1; mode=block
8 Set-Cookie: auth=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjp7I
9 Set-Cookie: auth_signature=-2_Fm9HD62XkeQ0C8jIHDE4Q5bAXB6gHeQCn-a9G
10 Content-Type: text/html; charset=utf-8
11 Content-Length: 2
12 ETag: W/"2-nOO9QiTIwXgNtWtBJezz8kv3SLc"
13 Date: Thu, 21 May 2020 14:45:20 GMT
14 Connection: close
15
16 OK
```

Search... 0 matches Pretty

146 of 1679616

Rysunek 16 Wynik przeprowadzonego ataku siłowego na hasło

Pomocnicze metody takie jak, resetowanie hasła powinny być równie dobrze zabezpieczone jak metoda uwierzytelniająca. Aplikacja realizuje odzyskiwanie haseł przez wysłanie tokenów na adres e-mail powiązany z kontem. Możliwy jest scenariusz, w którym atakujący rejestruje się w systemie, następnie resetuje hasło i ocenia, że token resetowania hasła zbudowany jest z 6 cyfr. Słaba złożoność tokenu umożliwia atakującemu przeprowadzenie ataku siłowego na token i odzyskanie hasła do konta innego użytkownika.

Podczas projektowania systemu należy zdefiniować wymagania dotyczące złożoności haseł oraz tokenów. W celu odpowiedniego określenia wymagań można posłużyć się wytycznymi przygotowanymi przez Natio-

nal Institute of Standards and Technology (NIST). Według zaleceń hasło powinno składać się z minimum 8 znaków, nie powinno być zbudowane z podciągu nazwy aplikacji. Hasło przed ustawieniem powinno być weryfikowane ze słownikiem najpopularniejszych haseł. Nie powinno stawiać się wymagań odnośnie występowania znaków specjalnych, natomiast dobrą praktyką jest umieszczenie miernika siły hasła [12]. Dodatkowym zaleceniem poprawiającym bezpieczeństwo w zakresie uwierzytelniania jest wprowadzenie podwójnej weryfikacji.

6.3. Problem z autoryzacją na poziomie funkcji

Atakujący w wyniku rekonesansu znalazł punkt końcowy, który powinien być udostępniany tylko administratorom, okazało się, że punkt końcowy przy wykorzystaniu metody GET nie implementuje kontroli autoryzacji. Każdy kto pozna adres zasobu, który nie wdraża autoryzacji może wysłać żądanie dzięki któremu otrzyma dane.

Znanym przykładem problemów z autoryzacją na poziomie funkcji jest podatność znaleziona w aplikacji wykorzystywanej do przeprowadzania głosowania w Izraelu [13]. Serwer zwracał odpowiedź zawierającą poufne dane bez konieczności autoryzacji użytkownika wysyłającego żądanie. Adres zasobu o postaci: „/get-admin-users”, został znaleziony w źródłach strony. Odpowiedź na żądanie pod znalezionym adresem zwracała dane administratorów systemu, część odpowiedzi została zaprezentowana na Rysunku 17.


```
96    {
97      "oid": 86,
98      "name": "הליכוד לכנסת",
99      "password": "[illegible]",
100     "email": "",
101     "campaignName": "כנסת 23 ליכוד",
102     "maxUsers": 10000,
103     "businessId": "",
104     "businessName": "",
105     "expirationDate": "19-01-2024 ",
106     "mainOfficeAddress": "",
107     "cost": null,
108     "purchasedSms": 710000,
109     "remainingSms": 340788,
110     "phone": "05[illegible]
111     "creationTime": "2020-01-19",
112     "usersCount": 100,
113     "adminId": "",
114     "allowExcelUpload": false,
115     "active": true,
116     "dataEncrypted": false
117   }
```


Rysunek 17 Poufne dane zwrócone bez wymaganej autoryzacji [13]

Ze względu na uporządkowanie API budowanego w architekturze REST odnalezienie punktów dostępów jest łatwym zadaniem nawet dla osób niedoświadczonych. Przed wdrożeniem aplikacji należy upewnić się, że wszystkie punkty końcowe zwracające poufne dane mają zaimplementowany i włączony mechanizm autoryzacji.

7. Wnioski

Na etapie projektowania powinny zostać określone wszystkie punkty końcowe wymagające uwierzytelnienia. Wdrażając mechanizmy uwierzytelnienia zaleca się korzystanie ze sprawdzonych standardów, podczas prób implementacji własnych rozwiązań istnieje wysokie prawdopodobieństwo popełnienia błędów. Punkty końcowe weryfikujące tożsamość użytkownika powinny implementować mechanizmy zabezpieczające przed atakami siłowymi, przykładem mechanizmu może być np. captcha. W systemie istnieje zazwyczaj kilka miejsc, są to miedzy innymi: resetowanie hasła, zmiana adresu e-mail, tak samo wrażliwych jak uwierzytelnianie, w tych miejscach warto wdrożyć podobne standardy. Jeżeli jest to możliwe warto zaimplementować mechanizm podwójnego uwierzytelnienia [14].

W celu ograniczenia problemów występujących podczas autoryzacji, należy zaprojektować proces z uwzględnieniem złożonej hierarchii użytkowników. Mechanizm autoryzacji powinien być wprowadzony na poziomie funkcji oraz każdego obiektu. Dobrą praktyką ograniczającą możliwości wysłania niepożądanych zapytań jest zastosowanie identyfikatorów GUID zamiast identyfikatorów, które pozwalają na odgadnięcie kolejnych wartości. Domyślnie przyznawane uprawnienia powinny być możliwie najmniejsze, uprawnienia należy zwiększać adekwatnie do potrzeb z wyraźną akceptacją przez uprawnione osoby. Mechanizm autoryzacji powinien być testowany automatycznie, testy powinny być wykonywane po każdej ingerencji w działanie mechanizmu [14].

Bezpieczna implementacja JWT możliwa jest wyłącznie przy pełnym zrozumieniu budowy i sposobów wykorzystania. Na etapie projektowania należy wybrać odpowiednie algorytmy podpisu, a w przypadku JWE także algorytmy szyfrowania. Podczas wdrożenia powinno się używać silnych kluczy kryptograficznych. Bezwzględnie należy zabezpieczyć system przed akceptacją algorytmu podpisu „none” i zwrócić szczególną uwagę na poprawność weryfikacji podpisu. Do wdrożenia JWT powinno używać się gotowych i sprawdzonych bibliotek, ponieważ własna implementacja jest narażona na błędy bezpieczeństwa. Przed wykorzystaniem biblioteki powinno zapoznać się z jej dokumentacją oraz wdrożyć monitorowanie wykrytych podatności i procedury aktualizacji. Poza ustawianiem krótkiego czasu ważności tokenów system powinien umożliwiać ręczne ich unieważnienie [1].

Należy zauważyć, że satysfakcjonujący poziom bezpieczeństwa można osiągnąć tylko przez spełnienie wszystkich wymogów bezpieczeństwa. Zaniedbanie któregokolwiek z nich powoduje powstanie luki, przez którą atakujący może uzyskać dostęp do danych lub funkcji.

8. Literatura

[1] Praca zbiorowa, Bezpieczeństwo aplikacji webowych, Securitum, Kraków 2019.

[2] J. S. Karsun, Solutions LLC Microservices API Security https://www.ijert.org/research/microservices-api-security-IJERTV7IS010137.pdf

[3] J. H. Saltzer, M. D. Schroeder, The Protection of Information in Computer Systems http://web.mit.edu/Saltzer/www/publications/protection/index.html

[4] OWASP Application Security Verification Standard, https://owasp.org/www-project-application-security-verification-standard/ [28.04.2020]

[5] OWASP Proactive Controls, https://owasp.org/www-project-proactive-controls/ [06.05.2020]

[6] Opis architektury REST, https://restfulapi.net/ [9.04.2020] .

[7] Porównanie popularność architektury REST i protokołu SOAP https://trends.google.com/trends/explore?date=all&q=REST%20API,%2Fm%2F077dn [18.04.2020].

[8] Specyfikcja JWT, https://tools.ietf.org/html/rfc7519 [11.04.2020].

[9] Informacje o JWT, https://jwt.io/introduction/ [11.04.2020]

[10] Opis biblioteki jsonwebtoken, https://www.npmjs.com/package/jsonwebtoken [11.04.2020].

[11] Narzędzie JWT cracker, https://github.com/brendan-rius/c-jwt-cracker [28.04.2020].

[12] Zalecenia dotyczące polityki bezpieczeństwa hasła https://pages.nist.gov/800-63-3/sp800-63b.html#sec5 [06.05.2020].

[13] Opis podatności związanej z brakiem uwierzytelniania https://dzone.com/articles/api-security-weekly-issue-70 [06.05.2020].

[14] Opis najbardziej krytycznych podatności API, https://owasp.org/www-project-api-security/ [11.04.2020].