REST API -kehitys tietoturvallisuusnäkökulmasta

Jani Karvo

Haaga-Helia ammattikorkeakoulu

Tradenomi

Amk-opinnäytetyö

2023

Tiivistelmä

Tekijä(t)
Jani Karvo

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
REST API -kehitys tietoturvallisuusnäkökulmasta

Sivu- ja liitesivumäärä
45 + 9

Tämä toiminnallinen opinnäytetyö käsittelee API:en kehitystä tietoturvallisuuden osalta, mitä asioita tulee huomioida, mitä riskejä on olemassa ja kuinka niiltä suojaudutaan. Työn osana toteutetaan Java-ohjelmointikielellä tehty tietoturvallinen API Spring-viitekehyksen puitteissa.

Opinnäytetyö sisältää teoria-, toiminnallisen, sekä pohdintaosuuden. Teoriaosuudessa kerätään tietopohja, jonka perusteella toiminnallinen osuus voidaan toteuttaa. Toiminnallisessa osuudessa kuvataan API:n kehitys ja siihen liittyvät valinnat. Lopuksi pohdinnassa arvioidaan työn tuloksia.

Teoriaosuus alkaa API:en historian ja perusteiden lyhyellä esittelyllä, jonka jälkeen edetään käsittelemään API:en tietoturvaa periaatteiden, suunnittelun ja uhkien tunnistamisen kautta. Seuraavaksi tehdään katsaus asiantuntijoiden listaukseen API:en kirjoitushetken kriittisimmistä tietoturvariskeistä. Lopuksi pureudutaan suojausmekanismeihin, joilla API turvataan. Teoriaosuudessa on tarkoitus luoda tietoperusta, jonka avulla voidaan kehittää tietoturvallinen API.

Toiminnallisessa osuudessa kuvataan API:n käytännön toteutus projektin kulun, teknologiavalintojen, sekä toteutettujen suojauksien ja testien osalta. API on toteutettu Spring Bootilla käyttäen apuna eri Spring-aliprojekteja. Tuloksena on API, joka on tietoperustan mukaisia parhaita käytänteitä vastaavalla tavalla suojattu yleisimpiä hyökkäyksiä ja uhkia vastaan.

Työn pääpaino on API:en tietoturvassa, eikä siinä perehdytä kattavasti ohjelmisto- tai API-kehityksen parhaisiin käytänteisiin, erilaisia mahdollisia suojausratkaisuja ei myöskään käydä laajasti läpi. API on toteutettu käyttäen suosittuja yleisimpiä teknologioita ja ratkaisuja, ja näin ollen esittelee vain yhden tavan tietoturvallisen API:n käytännön toteutukseen.

Asiasanat
API, REST, Tietoturva, Ohjelmointirajapinta, Java, Spring

Sisällys

1 Johdanto 1
2 REST API 4
  2.1 Perusteet 4
  2.2 Tietoturvallisuus 7
    2.2.1 Periaatteet 7
    2.2.2 Suunnittelu 8
    2.2.3 Trendit 9
    2.2.4 Uhkien tunnistaminen 10
3 API:en kriittisimmät tietoturvariskit 12
4 API:en suojausmekanismit 17
  4.1 Käytönrajoitus 17
  4.2 Autentikointi 18
  4.3 Valvontalokin kirjaus 19
  4.4 Auktorisointi 20
  4.5 Salaus 22
5 API:n suojaus käytännössä 24
  5.1 Käytetyt teknologiat 24
    5.1.1 Spring Boot 24
    5.1.2 Keycloak 25
  5.2 Projektista yleisesti 26
  5.3 Toteutetut suojaukset 27
    5.3.1 Käytönrajoitus 27
    5.3.2 Autentikointi 29
    5.3.3 Valvontalokin kirjaus 31
    5.3.4 Auktorisointi 32
    5.3.5 Salaus 34
    5.3.6 Muut suojaukset 35
6 Pohdinta 40
Lähteet 43
Liitteet 46
  Liite 1. Projektin rakenne 46
  Liite 2. RateLimitFilter-luokka 47
  Liite 3. SecurityConfig-luokka 48
  Liite 4. AuditLoggingFilter-luokka 49
  Liite 5. Autentikoinnin testit 50

Liite 6. Lokikirjauksen testit 51
Liite 7. Auktorisoinnin testit 52
Liite 8. Validoinnin testit 53
Liite 9. Spring Security HTTP-otsikot 54

1 Johdanto

Roy Fielding julkaisi 2000-luvun alussa tutkielmansa, jossa määritteli REST (Representational State Transfer) arkkitehtuurin. Samoihin aikoihin web-API:en (Application Programming Interface) pioneerit Salesforce, eBay ja Amazon julkaisivat ensimmäiset ohjelmointirajapintansa. Näiden tapahtumien johdolla modernit web-API:t alkoivat muotoutua ja niiden käyttö on siitä asti jatkuvasti kasvanut. Nykyään API:t ovatkin olennainen osa lukemattomia eri ohjelmistotuotteita ja -palveluita. (Lane 2019.) Organisaatiot käyttävät API:eja kaikenlaisen datan siirrossa ja eri palveluiden yhdistämisessä. Tästä syystä varsinkin arkaluonteista dataa käsittelevien API:en suunnittelussa ja kehityksessä tulisi erityisesti huomioida tietoturvallisuus, sillä puutteellisesti suojatut, rikkinäiset ja vaarantuneet API:t ovat usein tietomurtojen taustalla. (Red Hat 2019.)

Tämän opinnäytetyön tarkoituksena on perehtyä API:en turvallisuuteen liittyviin tekijöihin selvittäen, mitä kaikkea API:en kehityksessä tulee huomioida tietoturvan osalta, muun muassa uhkien, ongelmien ja toiminnallisten suosituksien suhteen. Työhön kuuluu myös API:n käytännön toteutus ja suojaus selvityksessä ilmenneiden parhaiden käytänteiden mukaisesti. Lopputuloksena on tarkoitus saavuttaa tietoturvallinen API, joka on tehokkaasti suojattu kaikkia kirjoitushetken yleisimpiä ja kriittisimpiä uhkia vastaan. API:en toteuttamiseen on useita erilaisia menetelmiä protokollien ja arkkitehtuurien suhteen, joista tämän työn toteutukseen on valittu REST-arkkitehtuuri. Tietoperustana työssä käytetään ajantasaista asiantuntijoiden laatimaa materiaalia, kuten kirjoja, artikkeleita ja tutkimuksia.

Valitsin työn aiheen, sillä tietoturva on erittäin tärkeä ja itseäni kiinnostava ohjelmistokehityksen alue, mutta joka usein jää ohjelmistojen liiketoiminnallisten ominaisuuksien varjoon. Ainakin oman kokemukseni mukaan tietoturvaa ja turvallisuutta edistävää ohjelmointia käsitellään suhteellisen vähän ottaen huomioon sen tärkeyden nykypäivänä. Suuristakin tietomurroista, jotka aiheuttavat paljon haittaa kaikille osapuolille, saa lukea uutisista tasaisin väliajoin. Tästä syystä haluan oppia aiheesta lisää yleisesti ja erityisesti API:en yhteydessä, koska API:t ovat nykyään hyvin yleinen osa ohjelmistoja, joita kasvavassa määrin käytetään julkisen verkon kautta, mikä puolestaan altistaa niitä tietoturvariskeille.

Opinnäytetyötä ohjaavat tutkimuskysymykset ovat: Mitä asioita tulee huomioida tietoturvallisen API:n kehityksessä? Kuinka tietoturvallinen API toteutetaan käytännössä?

API toteutetaan Java-ohjelmointikielellä Spring-viitekehyksen puitteissa, käyttäen Spring Bootia. Lisäksi apuna käytetään muita teknologioita ja työkaluja, kuten Keycloak, PostgreSQL ja Postman. Työssä käytetyt teknologiat on valittu niiden monipuolisuuden, yleisen suosion, sekä henkilökohtaisen preferenssin perusteella.

Työn tuotokset eivät sellaisenaan tule todelliseen käyttöön, koska työllä ei ole toimeksiantajaa. Pääasiallinen tarkoitus onkin oppia API:en tietoturvallisuudesta, sekä yleisesti tietoturvasta ja turvallisuuslähtöisestä ohjelmoinnista. Tästä syystä toteutettavan API:n perustoiminnallisuus ja liiketoimintalogiikka pidetään työssä minimalistisena pääpainon ollessa tietoturvaa parantavien ominaisuuksien toteutuksessa. Käytännössä API sisältää muutaman päätepisteen, jotka käyttävät yleisimpiä HTTP (Hypertext Transfer Protocol) -metodeja (GET, POST, PUT, PATCH, DELETE), joiden kautta käsitellään relaatiotietokannassa sijaitsevaa dataa.

Opinnäytetyö keskittyy REST API:en tietoturvallisuutta käsitteleviin kysymyksiin ja metodeihin, sekä siihen, miten turvallisuusominaisuudet toteutetaan käytännössä valituilla teknologioilla, kirjoitushetken parhaiden käytänteiden mukaisesti. Työn ei ole tarkoitus kattaa kaikkia yleisiä ohjelmisto- tai API-kehityksen parhaita käytänteitä, ja esimerkiksi epäsuorat tietoturvaa edistävät asiat, kuten API:n dokumentointi on rajattu työn ulkopuolelle. Myös oleelliset turvallisuuden osa-alueet, jotka eivät suoraan liity API:iin, kuten tiedon suojaaminen levossa on rajattu pois. Lisäksi apuna käytettyjä teknologioita, kuten Keycloakia, käsitellään vain niiltä osin, kun ne oleellisesti liittyvät API:n toimintaan tai on tarpeen toteutuksen kontekstin kannalta.

Työn tietoperustassa käsitellään aluksi REST API:en historiaa, toimintaperiaatteita ja arkkitehtuuria, jonka jälkeen edetään API:en tietoturvallisuuden periaatteisiin, trendeihin ja uhkien tunnistamiseen. Seuraavaksi käydään läpi tietoturva-ammattilaisten listausta API:en kriittisimmistä uhista ja riskeistä, ja viimeisenä esitetään suositeltuja käytännön suojausmekanismeja tietoturvauhkien neutralisoimiseksi. Empiria osiossa kuvataan toteutuksessa käytetyt teknologiat, projektin kulku, sekä API:iin implementoidut suojausmekanismit. Lopuksi tehdään yhteenveto koko opinnäytetyöprosessista, ja käydään läpi työn tavoitteiden toteutumista, siitä saatuja oppeja, sekä omaa kehittymistä.

Keskeiset käsitteet

API	Application Programming Interface (Ohjelmointirajapinta). Mekanismi, jonka kautta kaksi ohjelmistokomponenttia pystyvät kommunikoimaan keskenään protokollien ja määritelmien avulla (Lane 2022, 22–23).
CORS	Cross-Origin Resource Sharing. HTTP-otsikkopohjainen mekanismi, jolla palvelin voi ilmoittaa ne osoitteet, joista sen resurssien lataaminen on sallittu (MDN s.a.).
HTTP-otsikko	HTTP-pyynnön tai -vastauksen kenttä, joka välittää lisäkontekstia ja metatietoja pyynnöstä tai vastauksesta (IETF 2022).
JSON	JavaScript Object Notation. Avoimen standardin tiedostomuoto, joka käyttää avain-arvo-pareista muodostuvia objekteja tiedon tallentamiseen ja lähettämiseen (json.org s.a.).
JWT	JSON Web Token. Avoin RFC 7519-standardi, joka määrittelee tiiviin ja itsenäisen menetelmän tiedon siirtämiseksi turvallisesti osapuolten välillä JSON-objektina (Okta s.a.).
OAuth2	Auktorisointistandardi, joka on suunniteltu tarjoamaan verkkosivustolle tai sovellukselle pääsy muiden verkkosovellusten resursseihin käyttäjän puolesta (OAuth s.a.).
OIDC	OpenID Connect. OAuth 2.0 -protokollan päälle rakennettu identiteettikerros, jonka avulla voidaan varmistaa loppukäyttäjän henkilöllisyys auktorisointipalvelimen suorittaman todennuksen perusteella (OpenID s.a.).
REST	Representational State Transfer. Roy Fieldingin luoma arkkitehtuurityyli, joka kuvaa joukon periaatteita ja rajoitteita siitä, miten verkossa sijaitseva resurssi määritellään ja miten sitä käsitellään (Postman 2020).
Spring	Suosituin Java viitekehys, jonka tavoite on nopeuttaa ja helpottaa turvallisten Java-ohjelmistojen kehitystä (Spring s.a.).
TLS	Transport Layer Security. Salausprotokolla, joka on suunniteltu suojaamaan tietoliikennettä (Madden 2021, 78–79).

2 REST API

Modernit web-API:t, eli ohjelmointirajapinnat, alkoivat muotoutua 2000-luvun alussa, kun sosiaalisen median ja verkkokaupan palvelut kehittivät ja julkaisivat omia ohjelmointirajapintojaan, tavoitteenaan tehdä digitaalisista resursseistaan helpommin jaettavia ja saavutettavia (Lane 2022, 22–25). Modernien ohjelmointirajapintojen kehitystä edelleen vauhditti Roy Fieldingin esittelemä konsepti REST-arkkitehtuurista, jonka tarkoitus oli standardisoida web-ohjelmistoarkkitehtuuri ja helpottaa eri komponenttien välistä kommunikointia (Hawkins 2020).

REST-tyylisen ohjelmointirajapinnan pääasiallinen etu on standardisoitu arkkitehtuuri, jonka ansiosta itse ohjelmointirajapintaa tai sitä käyttävää asiakasohjelmaa ei ole sidottu tiettyyn alustaan, teknologiaan tai toteutustapaan (Microsoft 2022). Tästä syystä se on joustava ja skaalautuva ratkaisu, joka sallii ohjelmointirajapinnan täysin itsenäisen jatkuvan kehittämisen, ilman että sillä on vaikutusta sitä jo käyttävien asiakasohjelmistojen toimintaan (Postman 2020).

Ajan myötä organisaatioiden kehittäessä ohjelmointirajapintojaan kävi selväksi, että niiden kautta oli kätevä jakaa resursseja muiden organisaatioiden ja ohjelmistokehittäjien kanssa, ja näin laajentaa resurssien saavutettavuutta, sekä ajaa innovaatioita ja kehitystä (Hawkins 2020). Ohjelmointirajapintojen määrän kasvaessa ja yhä useamman järjestelmän sekä tehtävän tullessa niistä riippuvaiseksi, kasvavat myös riskit. Samat ominaisuudet, jotka tekevät rajapinnoista houkuttelevia ja helppokäyttöisiä vaihtoehtoja ohjelmistokehittäjille, avaavat myös mahdollisuuksia haitallisille toimijoille. (Madden 2021, 4.)

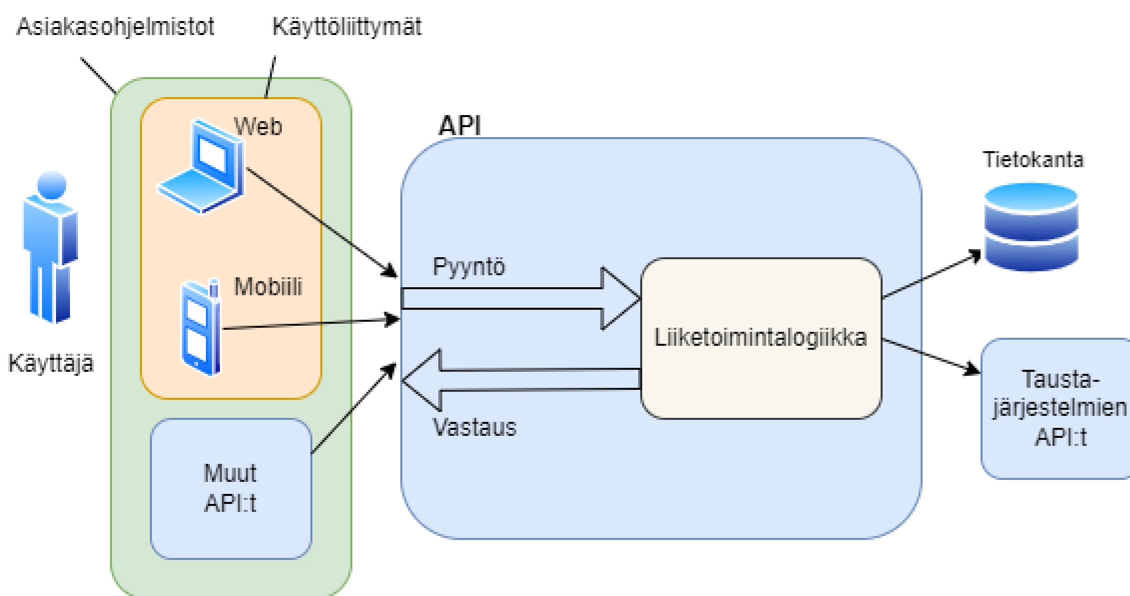
2.1 Perusteet

Ohjelmointirajapinnat juontavat juurensa automaattisen tietojenkäsittelyn alkuaikoihin. Ohjelmointirajapinta määrittelee, millä tavalla sen omaavan järjestelmän tai komponentin kanssa voidaan olla vuorovaikutuksessa, eli ohjelmointirajapinta toimii alla olevan järjestelmän käyttöliittymänä. Ohjelmointirajapintaa käyttävien järjestelmien ei tulisi olla millään tavalla tietoisia sen alla olevan järjestelmän sisäisestä toiminnasta ja kommunikoinnin tulisi tapahtua vain rajapinnan kautta. (Siriwardena 2020, luku 1.) Hyvin suunniteltua API:a pystyy käyttämään mikä tahansa asiakasohjelma, riippumatta esimerkiksi sen alustasta tai muusta teknologiasta. Lisäksi API:a tulisi pystyä edelleen kehittämään ilman että uudet ominaisuudet tai muutokset vaikuttavat API:a jo käyttävien asiakasohjelmistojen toimintaan millään tavalla. (Microsoft 2022.)

Ohjelmointirajapintojen kehittämiseen on olemassa useita eri tyylejä ja arkkitehtuureja, kuten SOAP, GraphQL tai RPC, sekä yleisin REST (Santos 2017; WebscrapingAPI 2021; Gutmann 2022). REST on arkkitehtuurityyli, joka perustuu hypermediaan ja toteutetaan yleensä HTTP-

protokollaa käyttäen. REST API:t suunnitellaan resurssien ympärille. Resurssi voi olla esimerkiksi mikä vain digitaalinen palvelu, objekti tai dataa. Jokaisella resurssilla on oma URI (Uniform Resource Identifior), eli yksilöllinen tunnus, jonka kautta resurssin kanssa voidaan kommunikoida. (Microsoft 2022.)

Kuten kuvasta 1 on havaittavissa, API voi palvella useita erilaisia asiakasohjelmistoja ja käyttöliittymiä. API prosessoi sille tulleet pyynnöt sisäisen toimintalogiikkansa mukaisesti ja palauttaa lopulta vastauksen. Riippuen API:n toimintalogiikasta ja sen toteutuksesta, voi API tarvittaessa ottaa käyttäjältä vastaan myös parametreja tai edelleen tehdä kyselyitä tietokantaan tai muille ohjelmointirajapinnoille.



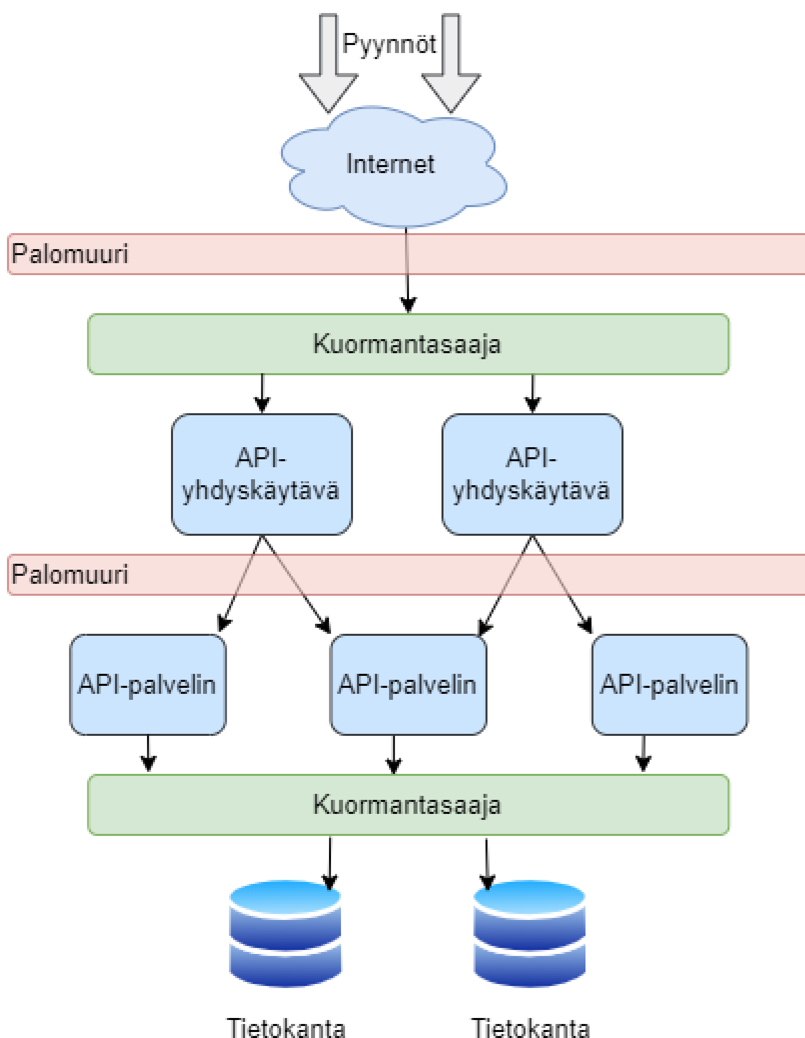
Kuva 1. API:n toimintalogiikka (mukaillen Madden 2021)

Kommunikoinnissa API:n kanssa URI-osoite määrittelee minkä resurssin kanssa ollaan tekemisissä, ja metodi määrittelee resurssille tehdyn pyynnön tyypin, eli miten resurssia käsitellään. HTTP-protokolla määrittelee useita metodeja, joista viisi yleisimmin käytössä olevaa ovat: (Postman 2020; Microsoft 2022.)

- GET-metodi, joka palauttaa pyynnön tekijälle osoitteessa sijaitsevan resurssin.
- POST-metodi yleensä luo uuden resurssin osoitteessa, mutta metodia voidaan käyttää myös muihin tehtäviin, jotka eivät välttämättä luo uusia resursseja.
- PUT-metodi luo, korvaa tai päivittää kokonaisuudessaan osoitteessa sijaitsevan resurssin.
- PATCH-metodi päivittää osoitteessa sijaitsevan resurssin osittain.
- DELETE-metodi poistaa osoitteessa sijaitsevan resurssin.

POST-, PUT- ja PATCH-metodien yhteyteen voidaan myös liittää parametreina tietoa tai ohjeita, joiden mukaisesti osoitteessa olevaa resurssia käsitellään (Postman 2020; Microsoft 2022). Yleisimmin käytetty dataformaatti HTTP-pyyntöjen ja -vastauksien yhteydessä REST API -arkkitehtuurissa on JSON (JavaScript Object Notation) (Jagger 2021). Olennainen osa REST-arkkitehtuurin mukaista kommunikointia on tilattomuus, eli jokaisen pyynnön tulee toimia itsenäisenä kokonaisuutena ja sisältää kaikki tarvittava informaatio pyynnön mukaisen operaation suorittamiseksi (Fielding 2000, 78–79).

Yleensä pyynnöt ohjelmointirajapintoihin liikkuvat usean tason läpi ennen kuin saavuttavat päämääränsä (kuva 2). TCP/IP-tasolta löytyy palomuuri, joka päästää läpi vain määritellyn liikenteen. Kuormantasaaja reitittää pyynnöt sopiville sisäisille palvelimille pyynnön tyypin ja palvelimien kuorman mukaisesti. API-yhdyskäytävä suorittaa API-palvelimen puolesta esimerkiksi laskennallisesti kuormittavia tehtäviä, kuten tunnistustietojen varmentamisen tai HTTPS-yhteyksien katkaisun. (Madden 2021, 11.)



Kuva 2. Verkkoliikenteen kulku API:lle (Mukaillen Madden 2021)

2.2 Tietoturvallisuus

Ohjelmointirajapintojen kautta asiakasohjelmat pääsevät käyttämään taustajärjestelmiä, mutta niiden avoimuus houkuttelee myös haitallisia toimijoita. API:en käyttö kasvattaa ohjelmistojen haavoittuvaa pinta-alaa, kun toiminnallisuutta siirtyy taustajärjestelmien sisältä asiakkaiden ulkoisille laitteille. Lisäksi HTTP-pyyntöjen tekeminen ohjelmointirajapinnoille altistaa pyyntöjen sisällön, kuten osoitteet, HTTP-otsikot, kyselyt ja parametrit näkyville, kasvattaen haavoittuvuuksien riskiä. (Hussain, Hussain, Noye & Sharieh 2020, 83.)

Puutteellinen ohjelmointirajapintojen turvaaminen voi johtaa tietomurtoihin ja sitä kautta kriittisten tietojen vuotamiseen tai menetykseen. Organisaatioille tietomurron seuraukset voivat olla esimerkiksi liiketoiminnan estyminen tai menetys, mikäli hyökkääjät saavat estettyä API:n toimintaa. Oikeudelliset ongelmat tai sanktiot, jos API:n suojaus ei ole vastannut viranomaisten vaatimuksia tai henkilötietoja on vuotanut. Kasvaneet operatiiviset kulut, jos hyökkääjät pääsevät ajamaan turhaa työtä taustajärjestelmissä tai maineen menetys, mikäli tieto organisaation tietovuodosta ajautuu julkisuuteen. (Geekflare 2022.)

Ohjelmointirajapintaturvallisuus sijoittuu laajempien sovellus-, verkko- ja tietoturva osa-alueiden risteykseen. Todellisuudessa ei ole olemassa täysin turvallista järjestelmää, eikä turvallisuudesta ole yhtä oikeaa määritelmää. Tämä juontuu siitä, että eri toimialoilla on erilaiset vaatimukset turvallisuuden suhteen, ja on mahdollista, että tietyn alan vakavakin turvallisuuspuute on olennainen osa toisen alan liiketoimintaa. API:n turvallisuutta suunniteltaessa tulee ottaa huomioon aina useita näkökulmia riippuen sen käyttökontekstista. Huomioon otettaviin asioihin kuuluvat muun muassa suojeltava omaisuus, kuten data, fyysiset laitteet ja muut resurssit, API:n toimintaympäristö ja siellä esiintyvät uhat, sekä organisaatiolle tärkeät turvallisuustavoitteet ja keinot niiden saavuttamiseksi. (Madden 2021, 8–9, 13.)

2.2.1 Periaatteet

Järjestelmien turvallisuustavoitteista on olemassa lukuisia standardeja ja periaatteita, joista yhtenä tunnetuimmista voidaan pitää CIA:ta (Confidentiality Integrity Availability), joka on lyhenne sanoista luottamuksellisuus, eheys ja saatavuus. Luottamuksellisuus, eli tietoon pääsee käsiksi vain ennalta määritellyt tahot. Eheys, eli tietoja ei ole mahdollista luoda, muuttaa tai tuhota luvattomasti. Saatavuus, eli API on saatavilla sen oikeutetuille käyttäjille esteittä aina kun he sitä tarvitsevat. (Siriwardena 2020, luku 2; Madden 2021, 14.) CIA:n yleisesti tärkeiden tavoitteiden lisäksi organisaatioilla voi olla myös muita toimialakohtaisesti tärkeitä tavoitteita, kuten vastuu ja kiistämättömyys (Madden 2021, 14). Periaatetta onkin myöhemmin jatkettu muotoon CIA-T (Traceability). Jäljitettävyys, eli tietojen muokkaamisesta tai käytöstä tulisi jäädä jälki, kuka teki, mitä ja milloin.

Jäljitettävyys, joka toteutuu yleensä lokitietojen säilyttämisenä on nykyään myös osa säädöksiä, kuten GDPR:ää (General Data Protection Regulation), joka vaatii tietojen kerääjiä pitämään henkilötietojen käsittelystä sekä käsittelijöistä lokikirjaa. (Deogun, Johnsson & Sawano 2019, 9.)

Toinen suositeltu ja yleinen lähestymistapa tietoturvan parantamiseen on noudattaa DiD (Defence in Depth) -periaatetta (Siriwardena 2020, luku 2). DiD tarkoittaa kerroksellista suojaustapaa, sillä mikään yksittäinen mekanismi tuskin riittää suojaukseksi kaikkia mahdollisia uhkia vastaan. Suojausmekanismien kerroksittaisen arkkitehtuurin on tarkoitus poistaa järjestelmän yksittäiset heikot kohdat, kasvattaa murtautumiseen tarvittavien resurssien määrää ja antaa aikaa huomata hyökkäys, ennen kuin kaikki suojaukset on läpäisty. (CIS 2023.)

Organisaatioiden tietoturvallisuustavoitteet voidaan rinnastaa vaatimusmäärittelyjen muihin ei-toiminnallisiin tavoitteisiin. Niille tyypillisellä tavalla voi olla hankalaa tai jopa mahdotonta varmentaa, että jokin tietty tavoite on täytetty, koska turvallisuusuhan olemattomuutta ei voida täydellisellä varmuudella todeta. Täydellisen varmuuden lisäksi myös riittävän turvallisuustason määrittäminen voi osoittautua hankalaksi tehtäväksi. Todellisuudessa onkin vain harvoin mahdollista estää kaikki mahdolliset hyökkäykset, eikä se välttämättä olisi taloudellisesti kannattavaakaan. Riippuen API:n kontekstista, jotkin uhat voi jättää jopa täysin huomiotta. Tärkeintä on tilanteen ja ympäristön realistinen arviointi, sekä keskittyminen todennäköisten ja olennaisten riskien torjuntaan. (Madden 2021, 14–16.)

2.2.2 Suunnittelu

Ohjelmiston kehitysvaiheessa kehittäjät yleensä keskittyvät ensisijaisesti liiketoiminnan vaatimusten mukaisen toiminnallisuuden toteuttamiseen ja muut vaatimukset, kuten tietoturvallisuus ovat toissijaisia. Odotetusti ohjelmiston tietoturvan taso vastaa läheisesti kehittäjien ammattitaidon ja kokemuksen tasoa. Näin ollen tietoturvan huomioiminen vasta kehitysvaiheessa vaatii, että jokainen kehittäjä on myös tietoturvan asiantuntija, mikä on sinänsä ongelmallista, koska kaikki eivät tähän rooliin pysty tai edes sitä halua. Siksi siirtämällä tietoturvan pääpaino ohjelmiston suunnitteluvaiheeseen saavutetaan johdonmukaisempi ja korkeatasoisempi lopputulos, eikä kehittäjien myöskään tarvitse jatkuvasti miettiä tietoturvallisuutta toiminnallisuuksien toteutuksessa. (Deogun ym. 2019, 14.)

Tietoturvan pääpainon siirtyessä suunnitteluun, saadaan turvallisuusnäkökulmista yhtenäisempi osa prosessia sen sijaan, että ne nähdään erillisinä lisävaatimuksia. Suunnitteluvaiheen hyvät arkkitehtuuri-, teknologia- ja muut rakenteelliset valinnat tekevät ohjelmoinnista automaattisesti tietoturvallisempaa, koska ne epäsuorasti estävät turvattomia toteutuksia. (Deogun ym. 2019, 18–20.) Lisäksi tietoturvallinen ohjelmistokehitys on yleisesti ottaen halvempaa sekä helpompaa, kun se

tehdään suunnittelun aikana odottamatta, että suojauksen tarve ilmenee myöhemmin kehitys- tai tuotantovaiheessa. Suojausmekanismien lisääminen jälkikäteenkin on yleensä mahdollista, mutta vaatii todennäköisesti paljon muutoksia, tehden siitä haastavaa ja kallista. (Madden 2021, 13.)

Yhtenä haastavimmista tietoturvan suunnittelun osista pidetään oikean suhteen löytämistä käyttäjäystävällisyyden ja suojausten välillä. Turvallisuusominaisuuksien ollessa liian monimutkaisia käyttäjät usein kiertävät ne jollain tavalla tai poistavat ne käytöstä kokonaan. Siksi suunnittelussa pitäisi pyrkiä siihen, että suojausten lisääminen ei saa tehdä järjestelmän käytöstä yhtään monimutkaisempaa. Olennaista on myös huomioida järjestelmän suorituskyky, koska suojausten lisääminen kasvattaa resurssien tarvetta, eli välillisiä kustannuksia tai odotusaikoja. (Siriwardena 2020, luku 2.)

2.2.3 Trendit

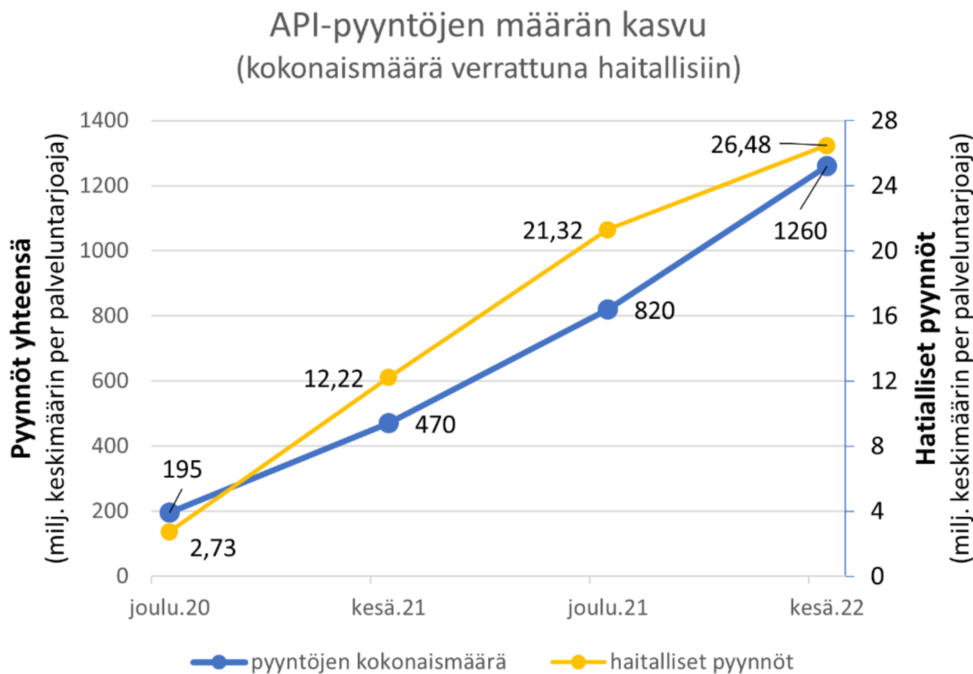
Järjestelmien verkottuneisuus, laajennettavuus, sekä monimutkaisuus ovat ensisijaiset trendit tietomurtojen määrän kasvun taustalla. Ohjelmointirajapinnat ovat keskeisessä osassa edistämässä verkottuneisuutta, mikä lisää ohjelmistojen yleisesti näkyvää pinta-alaa ja siten haavoittuvuutta. Suuri osa yritystason ohjelmistoista kehitetään laajennettavuusnäkökulma edellä. Mahdollisuus vain lisätä uusia ominaisuuksia ja toimintoja ilman muutoksia aiempaan koodiin tuo mukanaan hyötyjä, mutta kasvattaa samalla potentiaalista hyökkäyspinta-alaa. Monimutkaisuutta väistämättä luo jatkuvasti ja nopeasti kasvava järjestelmien, komponenttien ja integraatioiden määrä. (Siriwardena 2020, luku 2.) Tästä nopeasta kasvusta kertovat myös tutkimukset, kuten Salt:in (2022, 3) suorittama tutkimus, jonka mukaan heidän asiakasorganisaatioidensa keskimääräinen API:en määrä kasvoi 385 %:a vuoden 2020 viimeisen kvartaalin ja 2022 toisen kvartaalin välillä.

Tutkimuksissa on myös havaittu, että lähdekoodin monimutkaisuus aiheuttaa tietyn pisteen jälkeen ohjelmointivirheiden eksponentiaalisen kasvun. Siksi järjestelmien keskinäisen vuorovaikutuksen tulisikin olla mahdollisimman yksinkertaista, ymmärrettävää ja helposti implementoitavissa. Tämä vähentää ohjelmointivirheiden todennäköisyyttä, sekä helpottaa testausta ja ylläpitoa. (Siriwardena 2020, luku 2.)

Vaarassa eivät myöskään ole vain uusilla teknologioilla kehitetyt järjestelmät, vaan myös ne vanhemmat järjestelmät, jotka on kytketty verkkoon, mutta eivät välttämättä tue uusimpia suojausprotokollia. Lisäksi vanhoissa järjestelmissä käytetyt kirjastot voivat sisältää haavoittuvuuksia, joita ei ole korjattu muun muassa päivittämisen monimutkaisuuden takia. Puutteet järjestelmien suunnittelussa, integraatioissa tai ylläpidossa voivat helposti johtaa kokonaisten yhtenäisesti toimivien järjestelmäverkostojen vaarantumiseen. (Siriwardena 2020, luku 2.) API-

turvallisuustilannetutkimuksen mukaan 42 %:a vastanneista pitivätkin vanhentuneita ohjelmointirajapintoja suurimpana turvallisuusuhkana (Salt 2022, 6).

Salt:n (2023, 3) tutkimuksesta käy myös ilmi, että ohjelmointirajapinnoille tehtyjen haitallisten pyyntöjen määrä on noin 2 %:a pyyntöjen kokonaismäärästä, kuten kuvassa 3 esitetyistä tutkimus tuloksista on nähtävissä. Joulukuun 2020 ja kesäkuun 2022 välillä keskimääräinen palveluntarjoaja kohtainen pyyntöjen kokonaismäärä kasvoi 168 %:a ja haitallisten pyyntöjen määrä 117 %:a.



Kuva 3. API-pyyntöjen kokonaismäärän kasvu verrattuna haitallisten pyyntöjen määrän kasvuun (mukaillen Salt 2022)

2.2.4 Uhkien tunnistaminen

Hyökkäyksiä on olemassa laaja kirjo, mutta useimmat niistä voidaan lajitella muutaman tunnetun kategorian alle. Uhkien tunnistamisessa kannattaakin tunnistaa nämä laajemmat kategoriat, eikä yrittää määritellä yksittäisiä hyökkäyksiä. Uhkien mallinnuksessa voi käyttää apuna metodeja, kuten yleisesti tunnettua STRIDE:a. (Madden 2021, 18.)

STRIDE:n mukaiset kategoriat ovat (Madden 2021, 18; OWASP 2023a):

- Spoofing, esiintyminen toisena käyttäjänä.
- Tampering, datan, viestien tai asetusten muuttaminen haitallisessa tarkoituksessa.
- Repudiation, haitallisen teon kiistäminen, eli käytännössä hyökkäys järjestelmään, josta puuttuu lokitus tai lokien tuhoaminen tai väärennös hyökkääjän toimesta.

- Information disclosure, yksityiseksi tarkoitetun tiedon lukeminen tai paljastaminen oikeudetta.
- Denial of service, palvelun tai tiedonhaun estäminen muilta käyttäjiltä.
- Elevation of privilege, luvatta hankittu käyttöoikeus toiminnallisuuteen tai resurssiin.

Yleiset API-tietoturvan suojausmekanismit (käsitellään myöhemmin luvussa 4) ovat tehokas tapa suojautua kaikkia STRIDE:n määrittelemiä kategorioita vastaan. Nämä suojausmekanismit estävät useimmat yleisimmät hyökkäykset kokonaan tai vähintään huomattavasti lieventävät niistä aiheutuvia vaikutuksia. (Madden 2021, 18.)

3 API:en kriittisimmät tietoturvariskit

OWASP (Open Web Application Security Project) -yhdistyksen ylläpitämä API Security Top 10 -lista, kymmenestä kriittisimmästä riskistä ohjelmointirajapinnoissa on hyvä lähtökohta, kun suunnitellaan tai arvioidaan API:n tietoturvallisuutta (Lane 2022, 151–152). OWASP:ia pidetään web-applikaatioiden turvallisuuden perustana, mutta on tärkeää silti huomata, että näiltä riskeiltä suojautuminen ei vielä itsessään tee API:sta tietoturvallista. Riittävän tietoturvatason saavuttamiseksi ei ole olemassa yhtä yksinkertaista muistilistaa vaadittavista toimenpiteistä. (Madden 2021, 40.)

OWASP:in listaus on tulos monivaiheisesta arvioinnista ohjelmointirajapintakentän haavoittuvuuksista. Listaa varten on ensin kerätty, arvioitu ja kategorisoitu tietoa tapahtuneista tietomurroista julkisesti saatavilla olevista lähteistä. Seuraavaksi joukkoa tietoturva-ammattilaisia on pyydetty koostamaan omat kymmenen kriittisimmän riskin listansa. Lopuksi näiden kahden vaiheen tuloksista on luotu yhteenveto, jonka ohjelmointirajapintojen tietoturvaan erikoistuneista henkilöistä koostunut ryhmä on tarkastanut. (OWASP 2019, 30.) OWASP päivittää listauksiansa noin 3–4 vuoden välein, ja kirjoitushetken viimeisin API Security -listaus on vuodelta 2019.

Listauksessa jokainen uhka on arvioitu neljällä osa-alueella kolme portaisella asteikolla. Osa-alueet ovat hyväksikäytettävyyden helppous, haavoittuvuuden yleisyys, haavoittuvuuden havaittavuus ja hyökkäyksen seurauksien vakavuus. Asteikon tasot ovat helppo, keskiverto ja vaikea. Järjestys on laskeva, alkaen tärkeimmäksi koetusta. 2019 vuoden listauksen riskit ovat: puutteet objektitason auktorisoinnissa, puutteet käyttäjien tunnistamisessa, ylenmääräinen datan altistus, puutteet resursseissa ja käytönrajoituksessa, puutteet resurssitason auktorisoinnissa, kenttien joukkopäivitys, konfigurointivirheet, injektiot, puutteet omaisuudenhallinnassa, sekä riittämätön lokitus ja valvonta. (OWASP 2019, 6–7.)

Puutteet objektitason auktorisoinnissa

REST-tyylinen ohjelmointirajapinta on tilaton ja siten riippuvainen käyttäjien toimittamista parametreista, kuten objektin tunnuksesta, joka määrittelee, mitä objektia on tarkoitus käsitellä. Jos pääsynhallinta on puutteellista objektitasolla, voi hyökkääjille muodostua mahdollisuus hyväksikäyttää tätä mekanismia. Arkaluonteisten tietojen oikeudettomaan käyttöön voi esimerkiksi riittää vain objektin tunnuksen muuttaminen. Tästä syystä yleisesti ottaen on myös järkevää käyttää satunnaisia tunnuksia sarjojen sijasta, jotta niiden arvaaminen ei olisi helppoa. Lisäksi kaikkien ohjelmointirajapinnan päätepisteiden, jotka ottavat vastaan objektin tunnuksen ja suorittavat jonkun siihen kohdistuvan toiminnon, tulisi ensin tarkistaa onko pyynnön tehneellä käyttäjällä oikeuksia suorittaa kyseistä toimintoa kyseiselle objektille. (OWASP 2019, 8–9.)

Moderneissa järjestelmissä auktorisointimekanismit ovat monimutkaisia ja vaikka ohjelmisto sisältäisi oikean infrastruktuurin käyttäjän oikeuksien tarkistamiseen, eivät kehittäjät aina muista tai heillä ei ole riittäviä taitoja näiden mekanismien käyttöön. Pääsynhallinnan oikeaa toimintaa on myös tyypillisesti vaikea todentaa automaatiotestauksella. Puutteellinen objektitason pääsynhallinta on yleisin ohjelmointirajapintojen haavoittuvuus, joka yleensä johtaa oikeudettomaan tietojen käyttöön, muokkaamiseen tai poistamiseen. (OWASP 2019, 8–9.)

Puutteet käyttäjien tunnistamisessa

Myös käyttäjien tunnistaminen, eli autentikointi voi muodostua monimutkaiseksi kokonaisuudeksi ohjelmointirajapinnoissa. Tunnistusmekanismien rajoista sekä oikeasta toteutuksesta voi olla epätietoisuutta tai syntyä väärinkäsityksiä. Lisäksi autentikointimekanismi on usein julkinen ja siksi myös helppo kohde hyökkääjille. Puutteet, virheet tai käyttökohteeseen vääränlaisen autentikointimekanismin valinta voivat mahdollistaa hyökkääjille pääsyn muiden saman järjestelmän käyttäjien tileille. Autentikoinnin haavoittuvuuksia syntyy, jos esimerkiksi järjestelmä sallii heikot salasanat, ne tallennetaan salaamattomina palvelimelle, järjestelmä ei varmenna tokeneita tai niiden voimassa oloa, ei estä brute-force-hyökkäyksiä yksittäisiä käyttäjätilejä kohtaan tai tunnistautumisessa käytettäviä arkaluonteisia tietoja lähetetään URL-osoitteen ohessa. (OWASP 2019, 10–11.)

Autentikoinnin haastavuuden takia on suositeltavaa aina käyttää standardiksi muodostuneita metodeja, kirjastoja ja työkaluja, niin salasanojen tallennuksessa, tokenien luonnissa, kuin yleisessä autentikoinnin kulussakin. MFA (Multi-factor Authentication) tulisi olla käytössä aina kun se on mahdollista. Järjestelmän toteutukseen valittuihin teknologioihin tulisi perehtyä tarkasti, jotta ymmärretään mitä ne ovat ja kuinka niitä käytetään oikein. (OWASP 2019, 10–11.)

Ylenmääräinen datan altistus

Ohjelmointirajapintoja käytetään datan lähteenä ja usein kehittäjät yrittävät luoda yleismallisia ratkaisuja ilman, että API:n palauttaman datan arkaluonteisuutta on harkittu. Monesti luotetaan, että käyttöliittymä suorittaa datan suodattamisen ennen kuin se esitetään käyttäjälle. Objektien kenttien sisältämän datan arkaluonteisuutta pitäisi arvioida yksilöllisesti, sekä palauttaa vastauksessa vain oleellinen data. Liiallinen datamäärä pyyntöjen vastauksissa yleisesti johtaa arkaluonteisten tietojen vuotamiseen. Kyseisiä haavoittuvuuksia on useimmiten myös mahdotonta havaita automaatiotyökaluilla, koska datan erottelu ohjelmointirajapinnan käytön kannalta oikean ja tarpeellisen, sekä arkaluonteisen ja tarpeettoman välillä on hankalaa. (OWASP 2019, 12–13.)

Puutteet resursseissa ja käytönrajoituksessa

Pyynnöt ohjelmointirajapintoihin käyttävät resursseja, kuten muistia, tallennustilaa ja verkon kaistaa. Yksittäisen pyynnön tarvitsemat resurssit määrittyvät pitkälti päätepisteen toimintalogiikan, sekä käyttäjän syötteen mukaan, mutta myös muut samalle resurssille tehdyt pyynnöt kuluttavat resursseja. Resurssien käytön rajoitusten puutteesta ilmeneviä haavoittuvuuksia on helppo hyväksikäyttää yksinkertaisillakin pyynnöillä ilman autentikointivaatimuksia. Näiden DoS (Denial of Service) -hyökkäysten päämääränä on hidastaa tai kokonaan estää palvelun toiminta. (OWASP 2019, 14–15.)

Ohjelmointirajapinnat, joista kokonaan puuttuu käytönrajoitus tai sen toteutus on puutteellinen ovat suhteellisen yleisiä. API on haavoittuva, jos sen käyttöä ei ole rajoitettu oikealla tavalla muun muassa seuraavien osa-alueiden kohdalta: prosessin aikakatkaisu, käynnissä olevien prosessien määrä, yksittäisen käyttäjän tekemien pyyntöjen määrä, pyyntöjen mukana lähetetyn tietosisällön koko, yhden vastauksen palauttama tietomäärä tai allokoidun muistin määrä. (OWASP 2019, 14–15.)

Puutteet resurssitason auktorisoinnissa

Resurssitason auktorisointihaavoittuvuus ilmenee siten, että resurssille pyynnön tehneen käyttäjän oikeutta kyseisen pyynnön tekoon ei varmenneta. Tällaisen haavoittuvuuden hyödyntämiseksi hyökkääjän tarvitsee vain tehdä pyyntö suojaamattomaan päätepisteeseen. Lisäksi API:en kohdalla hyökkääjien mahdollisuuksia löytää kyseisiä haavoittuvuuksia edistää se, että API:en osoitteet ja metodit ovat yleensä jäsenneltyjä ja rakenteeltaan ennalta arvattavia. Esimerkiksi jos julkinen kirjan tietojen hakuosoite on /api/v1/kirja ja metodi GET, voi vain järjestelmänvalvojille tarkoitettu kirjan poisto-ominaisuus löytyä samasta osoitteesta vaihtamalla metodiksi DELETE. (OWASP 2019, 16–17.)

Oikeanlaisten auktorisointimekanismien implementointi saattaa muodostua haastavaksi, koska oikeuksien varmennus on mahdollista toteuttaa usealla eri tavalla, käyttäjillä voi olla useita rooleja, sekä organisaatiossa voi olla monimutkaisia hierarkiarakenteita. Hyvänä turvallisuuskäytäntönä pidetäänkin, että kaikki toiminnot ovat lähtökohtaisesti estetty ja toiminnon suorittaminen vaatii käyttäjältä nimenomaisen käyttöoikeuden tai roolin. (OWASP 2019, 16–17.)

Kenttien joukkopäivitys

Objektit pääsääntöisesti sisältävät useita kenttiä, joista osan pitäisi olla käyttäjän muokattavissa ja osan ei. Päätepiste on haavoittuvainen, jos käyttäjän on ilman asianmukaisia oikeuksia mahdollista muokata API:n sisäisten objektien kenttiä pyynnön mukana lähettämiensä parametrien mukaisesti.

Modernit viitekehykset tarjoavat kehittäjille usein valmiita funktioita, jotka automaattisesti sitovat käyttäjän lähettämät parametrit muuttujiksi ja sisäisiksi objekteiksi. Tästä syystä hyökkääjille voi avautua mahdollisuus muuttaa objektien kenttiä, joita kehittäjien ei todellisuudessa ollut tarkoitus altistaa muutoksille. Ilman oikeanlaista objektin kenttien tarkistusmekanismia hyökkääjä voi esimerkiksi oikean toiminnon lisäksi, kuten käyttäjänimen vaihdon ohessa, vaihtaa myös tilinsä oikeudet normaalista käyttäjästä järjestelmänvalvojaksi. (OWASP 2019, 18–19.)

Konfigurointivirheet

Virheelliset konfiguroinnit millä tahansa API-tietoturvan tasolla saattavat johtaa tietovuotoihin tai jopa kokonaisten järjestelmien vaarantumiseen. Tilanteeseen voi johtaa esimerkiksi TLS:n (Transport Layer Security) puuttuminen, väärin asetettu tai puuttuva CORS (Cross Origin Resource Sharing), ylimääräisten ominaisuuksien käytössäolo, viimeisimpien päivitysten puuttuminen tai liian tarkat virheilmoitukset API:n vastauksissa, jotka paljastavat tietoa järjestelmän sisäisestä toiminnasta. (OWASP 2019, 20–21.)

Käyttäjien syötteiden varmentamisen lisäksi tulisikin varmentaa myös API:n vastaukset. Varsinkin oletusasetuksien mukaiset virheilmoitukset voivat paljastaa liikaa tietoa järjestelmän sisäisestä toiminnasta. Hyökkääjän on helpompi etsiä tunnettuja haavoittuvuuksia, jos hän saa tietoonsa esimerkiksi järjestelmän käyttämää teknologiaa, versionumeroita tai sisäistä toimintalogiikkaa. Hyvä käytäntö on asettaa kaikille parametreille ja algoritmeille turvalliset oletusarvot, joita tulisi muuttaa vain viimeisenä keinona jonkun turvallisuuteen liittymättömän vaatimuksen saavuttamiseksi. (Madden 2021, 53, 74.)

Injektiot

Injektiohyökkäyksessä ohjelmointirajapinnalle syötetään komentoja minkä tahansa saatavilla olevan väylän läpi, kuten suorissa syötteissä tai parametreina. Hyökkääjä odottaa, että komennot menevät muuttumattomina ajettaviksi esimerkiksi SQL-kyselyihin tai komentoriville, ja suorittavat toimintoja, joihin heillä ei pitäisi olla oikeuksia. Injektiohaavoittuvuuksia syntyy, jos käyttäjien tai ulkoisten järjestelmien rajapinnalle lähettämiä syötteitä ei tarkasteta ja haitallista osaa datasta ei suodateta ennen sen käyttöä. Injektiot voivat saada aikaiseksi suurta haittaa tietovuotojen, tietojen menetyksen, palveluneston tai koko järjestelmän hallinnan menetyksen muodossa. (OWASP 2019, 22–23.)

Injektiohaavoittuvuuksia ilmenee usein, jos hyökkääjän syötteet toimivat tavalla, jota kehittäjä ei osannut odottaa ennalta. Tästä syystä syötteiden tarkistuksessa tulisi käyttää vakiintuneita kirjastoja ja formaatteja. (Madden 2021, 47.) Syötteiden tarkistuksessa kannattaa aina määritellä ennemminkin sallitut kuin estetyt syötteet, koska kaikkien haitallisten syötteiden ennakoiminen voi

olla haastavaa. Syötteitä ei tulisi myöskään ikinä sellaisenaan käyttää tietokantakyselyissä tai ajaa komentorivillä. Syötteiden tarkistuksen lisäksi haitallisten kyselyiden vaikutusta voidaan rajata esimerkiksi poistamalla API:lta valtuudet tiettyihin toimintoihin, kuten koko tietokannan tai yksittäisten taulujen poistamiseen. (Madden 2021, 43–44.)

Puutteet omaisuudenhallinnassa

Yhä käytössä olevat ohjelmointirajapintojen vanhat versiot ovat yleisesti ottaen heikommin suojattuja, kuin uudet uusimmalla teknologialla suojatut versiot. Vanhentuneet dokumentaatiot tai omaisuuden inventaario- tai käytöstäpoistostrategian puute tekee ohjelmistojen haavoittuvuuksien löytämisestä ja korjaamisesta haastavaa. Lisäksi modernit arkkitehtuurit, kuten mikropalvelut entisestään kasvattavat API:en lukumäärää, koska ne tekevät helpoksi uusien itsenäisten ohjelmistojen käyttöönoton, usein moninkertaistaen hallittavan omaisuuden määrän. Mikäli omaisuudenhallinnassa ei ole selkeää prosessia, kaikkia ohjelmistoja ei pidetä ajan tasalla ja vanhentuneita versioita ei poisteta käytöstä hallitusti, saattaa hyökkääjille avautua helppoja mahdollisuuksia päästä käsiksi arkaluonteisiin tietoihin. (OWASP 2019, 24–25.)

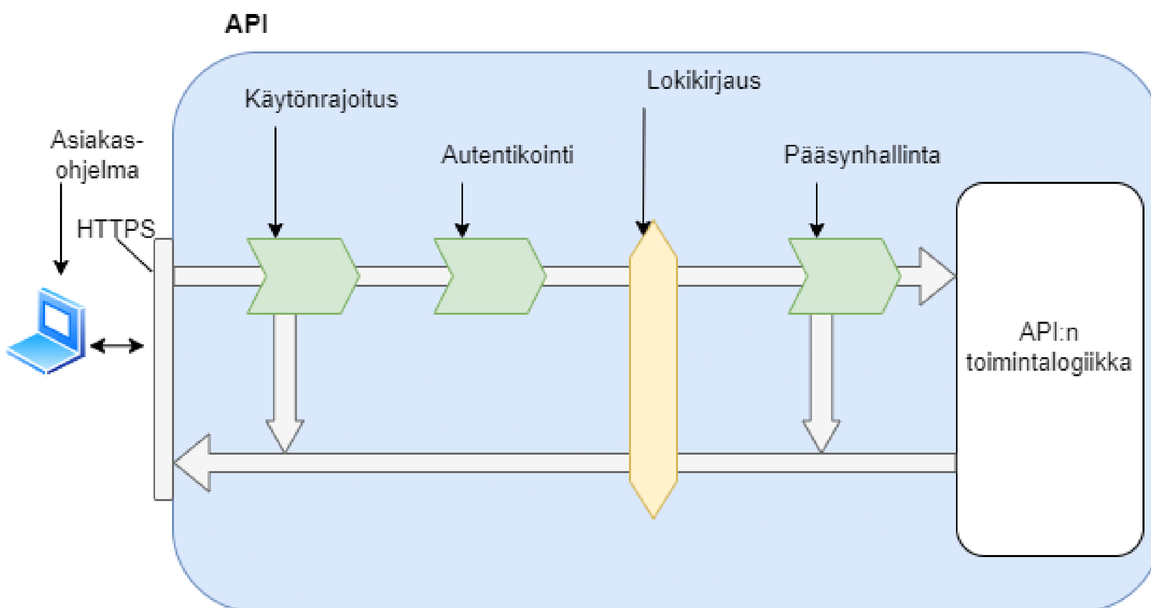
Riittämätön lokitus ja valvonta

Puutteellinen tai liian suppea valvonta ja lokien pito antaa hyökkääjille enemmän aikaa seurata ja väärinkäyttää järjestelmiä. Mikäli ohjelmointirajapintaan ei ole implementoitu riittäviä seurantamekanismeja, on lähes mahdotonta havaita epäilyttäviä tai haitallisia toimia niiden ollessa meneillään, tai jälkikäteen tutkinnassa selvittää tapahtumien kulkua tai vaikutusta. Ohjelmointirajapinnan käytöstä tulisi pitää käyttötarkoitukseen nähden riittävän tarkkaa lokikirjaa, lokitietojen eheys tulee taata, sekä lokitietojen ja rajapinnan infrastruktuurin tulee olla jatkuvan seurannan alaisena. Valvontaa varten kannattaa luoda tarkoituksenmukaisia koontinäyttöjä, sekä asettaa hälytyksiä epäilyttäville toiminnoille. (OWASP 2019, 26–27.) Automaatiotyökalujen ja koneoppimisen käyttö valvontalokien seurannassa ja analysoimisessa auttaa huomattavasti hyökkäyksien havaitsemisessa, sekä API:iin potentiaalisesti kohdistuvien uhkien mallintamisessa (Hussain ym. 2020, 85).

4 API:en suojausmekanismit

Uhkia neutralisoidaan implementoimalla suojausmekanismeja, joiden on käytännössä tarkoitus täyttää jokin järjestelmälle määritelty turvallisuustavoite. Hyvin suunnitelluissa ohjelmointirajapinnoissa yleisimmin käytössä olevat suojausmekanismit ovat salaus, autentikointi, pääsynhallinta, valvontalokien pito, sekä käytön rajoittaminen (kuva 4). Suojausmekanismit toteutetaan tyypillisesti DiD-periaatteen mukaisesti kerroksittaisena sarjana suodattimia, joiden läpi pyyntö kulkee ennen kuin se saavuttaa API:n ydinlogiikan. Suojausmekanismeja on mahdollista toteuttaa myös API:n ulkopuolisilla komponenteilla, kuten API-yhdyskäytävällä. (Madden 2021, 19.)

Ohjelmointirajapinnan tietoturvan tukemisessa tärkeinä elementteinä voidaan pitää myös muita epäsuorempia metodeja, kuten dokumentointia ja testausta. Lisäämällä tietoturvan keskeiset toiminnot osaksi dokumentaatiota autetaan käyttäjiä ymmärtämään kokonaisuutta sekä käyttämään ohjelmointirajapintaa oikein. Automatisoitu testaus taas on tehokas tapa todentaa API:n olevan yhä suojattu yleisimpiä uhkia vastaan muutosten yhteydessä. (Lane 2022, 150.)



Kuva 4. API:n yleisimmät suojausmekanismit (mukaillen Madden 2021)

4.1 Käytönrajoitus

Palveluiden saatavuuteen kohdistuvien DoS-hyökkäyksien tavoitteena on ylikuormittaa ohjelmointirajapinta tai jokin muu verkossa sijaitseva resurssi lähettämällä sille turhia pyyntöjä, joiden takia palvelu hidastuu huomattavasti tai kaatuu, eikä enää pysty palvelemaan sen oikeita käyttäjiä (IBM 2023). Tämän kaltaisia hyökkäyksiä on vaikea täysin estää, koska ne usein toteutetaan kaapatuilla resursseilla, joten hyökkääjä pystyy luomaan palveluun paljon ylimääräistä verkkoliikennettä ilman

suuria kustannuksia itselleen. Hyökkäysten torjuminen kuitenkin voi vaatia paljon resursseja sekä aikaa. (Madden 2021, 64.) Käytön rajoittamisen periaate on yksinkertainen: API-palvelimen resurssit ovat rajalliset, joten API:n käyttöä on myös rajattava, jotta sen saatavuus ei kärsi. Turvallisuuden lisäksi käytönrajoitus parantaa API:n laatua. API:t ovat yleensä useiden eri tahojen käytössä, ja yksittäisiä oikeitakin käyttäjiä tulee rajoittaa, jotta API:n saatavuus pysyy kaikille käyttäjille tasapuolisena. (DataDome 2020.)

Useimmiten DoS-hyökkäykset rajapintoihin ovat autentikoimattomia pyyntöjä, joten yksi mahdollisuus olisi estää kaikki autentikoimattomat pyynnöt resurssille. Autentikointimekanismit kuitenkin usein kuluttavat suhteellisen paljon resursseja, joten se ei yksistään riitä DoS-hyökkäyksien suojaksi. Autentikoimattomien pyyntöjen ei tulisi koskaan pystyä kuormittamaan huomattavan suurta määrää palvelimen resursseja, joten käytönrajoitusmekanismien tulisi olla ensimmäinen suojaus, jonka läpi rajapinnalle tehty pyyntö kulkee. (Madden 2021, 65.)

Optimaalisessa tilanteessa haitallista liikennettä rajoitetaan tai se estetään aikaisin, esimerkiksi API-yhdyskäytävän tai kuormantasaajan tasolla varmistaen, että yksittäiset API-palvelimet eivät ylikuormitu. Hyvä käytäntö on rajoittaa liikennettä silti yksittäistenkin palvelimien kohdalla suojaustason parantamiseksi. (Madden 2021, 66–67.) Rajoitus voidaan toteuttaa monella tapaa, muun muassa käyttäjäkohtaisesti tai API:n kokonaisuudessa prosessoimien pyyntöjen osalta. Käytännössä rajoitus voi olla esimerkiksi pyyntöjen määrän rajaus, kuten 100 pyyntöä sekunnissa, jonka ylittävät pyynnöt estetään, tai sallimalla pyynnöt vain tietyistä IP (Internet Protocol) -osoitteista. Lisäksi on olemassa dynaamisia algoritmipohjaisia rajoituksia, jotka jatkuvasti tarkkailevat API:n käyttöä ja tekevät rajoituspäätöksiä tilannekohtaisesti. (DataDome 2020.)

4.2 Autentikointi

Rajoitusmekanismin jälkeen seuraava suojaustaso on käyttäjän autentikointi, koska pääsääntöisesti kaikki muut suojaukset, kuten lokitus ja auktorisointi tarvitsevat tiedon siitä, kuka pyynnön on tehnyt. Autentikoinnin suhteen on tärkeä huomata, että autentikointimekanismi ei ikinä hylkää pyyntöjä suoraan, vaikka käyttäjän tunnistaminen epäonnistuisi, vaan päätös autentikoinnin tarpeesta tapahtuu pääsynhallintatasolla. Tämä johtuu siitä, että jokainen pyyntö ei välttämättä vaadi käyttäjän tuntemista, ja valvonnan kannalta on olennaista, että pyyntöjä ei hylätä ennen niiden kirjautumista valvontalokiin. (Madden 2021, 70.)

Käyttäjien autentikoimiseen on useita eri metodeja ja niiden yhdistelmiä, kuten MFA, biometrinen ja salasana- tai token-pohjainen (EDUCBA 2023). Yksi käytetyimmistä mekanismeista on HTTP-perusautentikointi, jossa käyttäjältä pyydetään käyttäjänimi sekä salasana, jotka koodattuna liitetään pyynnön HTTP-otsikoksi tunnistautumista varten. Tätä metodia tulisi käyttää vain TLS-yhteyttä

käyttävissä pyynnöissä, koska perusautentikoinnin purkaminen on suhteellisen helppoa kaikille, jotka pystyvät verkkoliikennettä kuuntelemaan. (Madden 2021, 71.) HTTP-perusautentikointi on kevyt ja helppokäyttöinen mekanismi, mutta sen arkkitehtuurin sisältämien turvallisuusriskien takia sopii käytettäväksi lähinnä sisäisissä verkoissa (Sandoval 2018).

Token-pohjainen autentikointi puolestaan on hallitseva mekanismi ohjelmointirajapintakäytössä ja kattaa useita erityylisiä toteutusratkaisuja, jotka soveltuvat erilaisiin tilanteisiin. Token-autentikoinnin idea on, että käyttäjä esittää tunnistetietonsa kerran ja saa vastineeksi tokenin, jota voi käyttää API:lle tehtyjen pyyntöjen autentikoinnissa tokenin voimassaolon ajan. (Madden 2021, 99, 102.) Modernin token-pohjaisen autentikoinnin etu on tilattomuus. API-palvelimen ei tarvitse säilyttää tietoja kirjautuneista käyttäjistä ja istunnoista, vaan jokaisen pyynnön tekijä tunnistetaan tokenista, jonka aitous voidaan varmentaa sen liikkeelle laskeneelta taholta. (Noroff 2023.)

Yksi käytetyimmistä token-pohjaisista toteutustavoista on OAuth 2.0 -protokollan päälle rakennettu autentikointikerros OIDC (OpenID Connect). Protokollan mukaisesti käyttäjä tunnistautuu erillisen auktorisointipalvelimen kautta ja saa käyttöönsä tokenin, jota voi käyttää tunnisteena. Token lisätään API:lle tehtyjen pyyntöjen yhteyteen ja API tarkistaa tokenin aitouden sen liikkeelle laskeneelta auktorisointipalvelimelta. OIDC:n mukaiset tokenit noudattavat JWT (JSON Web Token) -standardia ja voivat sisältää käyttäjätietojen lisäksi paljon muutakin informaatiota. (Levin 2019.)

Token-pohjainen autentikointi on yleistynyt osittain juuri JWT-standardin myötä. Standardisoinnin ansiosta JWT:t ovat yhteensopivia useimpien palveluiden, sekä ohjelmointityökalujen ja -kirjastojen kanssa. Useimmat API-viitekehykset tukevat JWT:tä ja lähes jokaiselle ohjelmointikielelle on JWT-kirjastoja. (Madden 2021, 184.) JWT on itsenäinen formaatti, jolla tietoa voi turvallisesti siirtää osapuolten välillä, ne ovat digitaalisesti allekirjoitettuja, joten tunnistetietoihin voidaan luottaa, sillä niiden sisällön eheys on helposti tarkastettavissa. JWT on JSON-objekti, joka koostuu kolmesta osasta, jotka ovat otsikko, tietosisältö ja signeeraus. Otsikko sisältää tyypillisesti tiedot tokenin tyypistä ja allekirjoitusalgoritmista. Tietosisältöön kuuluvat esimerkiksi tokenin vanhenemisaika, liikkeellelaskija ja käyttäjän tai muun entiteetin tietoja. Signeeraus on tunniste, joka luodaan tokenin otsikon, tietosisällön, sekä salausavaimen yhdistelmästä. Signeeraus mahdollistaa tokenin eheyden varmentamisen. (Okta s.a.)

4.3 Valvontalokin kirjaus

Vastuun valvonta on riippuvainen tiedoista, kuka teki, mitä ja milloin. API:en kohdalla helpoin ratkaisu näiden tietojen keräämiseen on valvontalokin kirjaaminen kaikista käyttäjien tekemistä toiminnoista. Lokituksen tulisi tapahtua heti autentikoinnin jälkeen ennen auktorisointimekanismia, joka voi hylätä API:lle tehdyn pyynnön. Hylättyjen pyyntöjen tallentuminen lokitietoihin on tärkeää,

koska hylätyt pyynnöt voivat olla merkki hyökkäysyrityksestä. Lokikirjauksen tulisi tapahtua sekä ennen, että jälkeen päätepisteen operaation, jotta on mahdollista todentaa mitä pyyntöjä oli käsitellyssä, mikäli prosessi kaatuu kesken kaiken. Valvontalokit ovat erittäin tärkeä osa tietoturvallisuutta, ja ne tulisi säilyttää kestävässä tallennusmuodossa esimerkiksi tietokannassa, jotta niitä ei menetetä, vaikka API-palvelin kaatuisi. Tuotantoympäristössä lokit yleensä lähetetään keskitettyyn säilöön, jossa SIEM (Security Information and Event Management) -työkaluilla voidaan analysoida kokonaisuutta poikkeavuuksien ja uhkien varalta. (Madden 2021, 82–84.)

Hyvästä lokikirjauksesta käy ilmi vähintään tapahtuman ajankohta, tekijä, käyttöoikeuden taso, pyynnön alkuperä sekä toiminto ja sen lopputulos. Lokituksen määrä tulee arvioida aina tarpeen mukaan. Liian suuri tietomäärä voi tukkia järjestelmän ja vaikeuttaa hyödyllisen tiedon etsimistä massasta. Lokitiedot sisältävät usein myös henkilötietoja tai muita arkaluonteisia tietoja, joten niiden säilyttämisessä on otettava huomioon tietoturva ja mahdolliset säädökset, kuten GDPR. Yleensä pääsy lokitietoihin tulisi rajata vain pienelle määrälle luotettuja käyttäjiä tai tarkastajia. Tehtävien eriyttämisperiaatteen mukaisesti yhden tahon ei pitäisi vastata useista eri vastuullisista tehtävistä. Esimerkiksi järjestelmänvalvojien ei tulisi hallita valvontalokeja virheiden ja väärinkäytön välttämiseksi. (Kyberturvallisuuskeskus 2023.)

4.4 Auktorisointi

Auktorisointi tapahtuu autentikoinnin ja ensimmäisen lokikirjauksen jälkeen. Mikäli käyttäjällä on oikeus pyynnön tekoon, voidaan API:n toimintalogiikan koodi ajaa. Käyttöoikeuden puuttuessa pyyntö tulee hylätä välittömästi ilman että yhtään toimintalogiikan koodia on ajettu. Tilanteissa, joissa käyttäjän pyyntö hylätään, tulee hänelle palauttaa virheilmoitus, jossa hylkäyksen syy on eritelty. Yksinkertaisin käyttöoikeuden tarkistusmekanismi, jolla varmistetaan, että vain oikeutetut käyttäjät voivat tehdä pyyntöjä API:lle, on vaatia kaikkia käyttäjiä tunnistautumaan. Usein kuitenkin tarvitaan monimutkaisempaa logiikkaa, koska harvoin kaikki käyttäjät tarvitsevat kaikkia ominaisuuksia tai toiminnallisuuksia. Vähimpien oikeuksien periaatteen mukaisesti käyttäjien oikeudet tulisi rajata vain niihin toimintoihin, joita he jatkuvassa toiminnassaan tarvitsevat. (Madden 2021, 88–90.)

Käyttöoikeuksien hallinta

Käyttöoikeuksien hallintaan on olemassa laaja valikoima erilaisia ratkaisuja ja tyylejä. Tärkeintä on kuitenkin tunnistaa tarve hallinnollisille käytännöille, sekä valita liiketoiminnan tarpeisiin sopiva hallintomalli. Käytännössä käyttöoikeuksien hallintalogiikka voi olla esimerkiksi resurssille määritelty lista sallituista käyttäjistä tai rajoitus pyynnön lähteen, kuten IP-osoitteen perusteella, tai useampien tekijöiden yhdistelmä. (Microsoft 2023.)

Karkeasti käyttöoikeuksienhallinta voidaan jakaa kahteen malliin, RBAC (Role-Based Access Control) ja ABAC (Attribute-Based Access Control), joilla on omat vahvuutensa. RBAC on roolipohjainen tyyli, jossa roolit määräytyvät liiketoimintalogiikan mukaisesti. Käyttäjälle suoraan määriteltyjen oikeuksien sijasta hänelle määritellään rooli, ja jokainen rooli omaa tietyn tason käyttöoikeudet. ABAC on attribuuttipohjainen tyyli, jossa käyttöoikeus johonkin resurssiin määräytyy tilanteen attribuuttien mukaisesti. Esimerkiksi oikeus johonkin resurssiin voi määräytyä kellon ajan, käyttäjän organisaation ja pyynnön alkuperän mukaan. Yleisesti ottaen RBAC-malli on yksinkertaisempi ja helppokäyttöisempi, kun taas ABAC-malli on joustavampi ja yksityiskohtaisempi. (AWS 2023.)

Käyttöoikeuksien tarkastus implementoidaan usein suoraan API:n resurssin liiketoimintalogiikan yhteyteen, koska käyttöoikeuksien määritys on viime kädessä liiketoiminnallinen kysymys. Toisaalta käyttöoikeuksien tarkastuksen erottaminen omaksi toiminnokseen helpottaa niiden keskitettyä hallintaa, varsinkin mahdollisten muutosten yhteydessä, sekä varmistaa niiden johdonmukaisen soveltamisen. (Madden 2021, 91.)

OAuth2 ja OIDC

Organisaatiot kasvavassa määrin avaavat API:nsa myös kolmansien osapuolten käyttöön. Aiemmin tällainen toiminta on vaatinut API:n käyttöön oikeuttavan käyttäjätunnuksen ja salasanan luovuttamista, jotta kolmas osapuoli pystyy hakemaan tietoa API:n kautta. Tunnusten jako toiselle tai useille muille osapuolille kasvattaa kuitenkin huomattavasti turvallisuusriskiä. Token-pohjainen autentikointi osaltaan ratkaisee kyseisen ongelman, koska tunnusten sijaan kolmannelle osapuolelle annetaan käyttöön aikarajoitettu token, joka oikeuttaa API:n käyttöön. Tokenin sisältämät oikeudet voidaan rajata vain tarvittavaan toimintoon ja ne voidaan kumota, kun pääsylle ei ole enää edellytyksiä. (Siriwardena 2020, luku 4; Madden 2021, 219–220.)

Yleisin käytössä oleva protokolla oikeuksien delegointiin kolmansille osapuolille on OAuth 2.0. Yhdessä OIDC:n kanssa nämä protokollat määrittelevät standardisoidun keskitetyn tavan token-pohjaiseen autentikointiin ja auktorisointiin. Keskitetty autentikointi mahdollistaa myös SSO (Single Sign-On) -järjestelmän, jonka avulla käyttäjät voivat tunnistautua useaan eri API:iin ja palveluun samoilla tunnuksilla. (Madden 2021, 217.)

On huomattava, että OAuth 2.0 ja OIDC käsittelevät eri aiheita, eli OAuth 2.0 delegoitua auktorisointia ja OIDC autentikointia. OAuth 2.0 -protokollan mukainen token on vain käyttöoikeus johonkin resurssiin, eikä kerro käyttäjän identiteetistä. Mikäli käyttäjän identiteetti halutaan myös varmentaa, tulee käyttää OIDC-protokollaa, joka on OAuth 2.0:n päälle rakennettu identiteettikerros. Kun käyttäjä tunnistautuu OIDC:ta tukevalla OAuth 2.0 -auktorisointipalvelimella, palauttaa palvelin tokenin, joka sisältää käyttäjän identiteetin sekä käyttöoikeudet. (Siriwardena 2020, luku 4.)

Tyypilliseen OAuth 2.0 -protokollan mukaiseen todennuskulkuun kuuluu neljä osapuolta: Resurssin omistaja, jonka resurssi voi olla esimerkiksi käyttäjätili jollain alustalla. Resurssipalvelin, kuten mainitun alustan API, jonka käyttö vaatii tietyn oikeuden. Asiakasohjelmisto, joka haluaa oikeuden API:n päätepisteen käyttöön resurssin omistajan puolesta. Auktorisointipalvelin, jonka kautta asiakasohjelmisto saa resurssin käyttöön oikeuttavan tokenin. Käytännön esimerkkinä (havainnollistettu kuvassa 5) käyttäjä vierailee kolmannen osapuolen ohjelmistossa, kuten web-sovelluksessa ja haluaa tehdä jonkun toiminnon, joka vaatii hänen omistamansa resurssin käyttöä. Web-sovellus ohjaa käyttäjän tunnistautumaan auktorisointipalvelimelle, joka tunnistaa käyttäjän ja palauttaa toimintoon oikeuttavan tokenin web-sovellukselle. Web-sovelluksen saama token oikeuttaa sen suorittamaan toiminnon käyttäjän omistamalle resurssille. (Siriwardena 2020, luku 4.)

Kuva 5. Tyypillinen OAuth 2.0 -protokollan mukainen todennuskulku

4.5 Salaus

Autentikointi estää käyttäjien oikeudettomat pyynnöt API:lle ja siten suojelee tietoja ja prosesseja. Arkaluonteisten tietojen suojaksi tarvitaan kuitenkin myös salausta. Mikäli dataa API:lle lähetetään ja vastaanotetaan salaamattomana verkon yli, voi kuka tahansa samassa verkossa oleva lukea viestien sisältöä. API:en tulisi sallia vain HTTP:n laajennusta käyttävä HTTPS-liikenne (Hypertext Transfer Protocol Secure), jossa TLS-protokolla mahdollistaa salatun verkkoliikenteen osapuolten välillä. (Madden 2021, 78–82.) TLS:llä salattua kommunikaatiota kuunnellessaan potentiaalinen hyökkääjä pystyy ainoastaan toteamaan, että kaksi tahoa kommunikoivat keskenään. Hän ei pysty lukemaan salattua sisältöä tai muuttamaan viestien sisältöä ilman että osapuolet sen huomaavat.

On kuitenkin huomattava, että TLS salaa viestien sisällön vain sen verkon ylitse kulkeman matkan ajan, ja päämäärän saavuttaessaan salaus puretaan. (Aumasson 2018, luku 13.)

Token-pohjaisessa autentikoinnissa asiakasohjelma tallentaa tokenit muistiin, eli yleensä selaimen evästeisiin tai istunnon tallennustilaan. Tämä altistaa tokenit useille erityylisille hyökkäyksille, kuten kalastelulle tai XSS:lle (Cross Site Scripting), eli tokenin sisältämät käyttäjän henkilökohtaiset tiedot voivat olla vaarassa. Lisäksi tokenit voivat sisältää API:n toimintaan liittyviä yksityiskohtia, joita ei haluta paljastaa edes käyttäjälle, eli ainakin tokenin sisällön osittainen salaus voi olla haluttavaa. On myös huomattava, että monet salausalgoritmit varmistavat vain tietojen luottamuksellisuuden, eivät eheyttä, joten hyökkääjät eivät pysty lukemaan tietoja, mutta niiden muokkaaminen voi olla mahdollista. (Madden 2021, 195–198.) AE (Authenticated Encryption) -algoritmit takaavat sekä viestien luottamuksellisuuden että eheyden. Turvallisimmat AE-algoritmit ensin salaavat viestin sisällön, jonka jälkeen luovat siitä MAC:n (Message Authentication Code), jonka avulla viestin eheys voidaan todentaa. (Aumasson 2018, luku 13.)

JWT:t ovat yleensä vain signeerattuja, eli eheys on varmistettu, mutta ei luottamuksellisuutta. JWT on mahdollista myös salata. Salaus kuitenkin kuluttaa suhteellisen paljon resursseja, ja lisäksi salausmekanismit ovat haastavia konfiguroida ja ylläpitää. Parhaana käytäntönä pidetäänkin kaikkien arkaluonteisten tietokenttien poistamista tokenista ja tarvittaessa näiden tietojen noutamista erillisestä lähteestä. JWT:t ovatkin parhaimmillaan autentikointi ja auktorisointi tarkoituksessa ilman että sisältävät arkaluonteisia tietoja käyttäjästä tai API:sta. (Curity 2023.)

5 API:n suojaus käytännössä

Tässä luvussa käsitellään opinnäytetyön osana suoritettua tietoturvallisen API:n käytännön toteutusta sekä projektissa käytettyjä teknologioita. Lopputulos on saavutettu huomioiden edellisissä luvuissa kuvatut perusteet, tietoturvariskit sekä suojausmekanismit, ja toteuttamalla niitä vastaavat suojausratkaisut. Toteutetun API:n kautta voi hakea ja muokata relaatiotietokannassa sijaitsevaa dataa kirjoista. API ei tule todelliseen käyttöön, joten sen sisältämää toiminnallisuutta ja liiketoimintalogiikkaa on toteutettu vain se määrä, joka on välttämätön suojausmekanismien demonstroimiseksi.

Tuotoksella ei ole erityistä kohderyhmää ja se toteutetaan käytännössä oman oppimisen tukemiseksi, jotta teoriatason tiedon lisäksi kertyy kokemusta myös siitä, kuinka ohjelmisto käytännössä suojataan. Laadullisten mittareiden suhteen lopputulosta voidaan pitää onnistuneena, mikäli API on onnistuttu suojaamaan vähintään yleisimpiä hyökkäyksiä vastaan, huomioon ottaen myös OWASP:in listauksen riskit ja ohjeistukset.

Luvussa kuvaillaan mitä projektin aikana on tehty, samalla havainnollistaen tehtyjä ratkaisuja rajatuilla koodilohkoilla. API:n keskeisimpien luokkien lyhentämätön lähdekoodi löytyy lisäksi dokumentin liitteistä ja projektin koko lähdekoodi löytyy osoitteesta: https://github.com/janijk/thesis-api.

5.1 Käytetyt teknologiat

Projektin toteutuksen kannalta tärkein teknologia on Spring, jonka puitteissa API kehitetään, ja joka vaikuttaa eniten lopputuloksen arkkitehtuuriin, ratkaisuihin ja toiminnallisuuksiin. Kokonaisuuden ja yleisen toiminnan kannalta toinen huomattava teknologia on Keycloak, jonka kautta käyttäjä tunnistautuu ja saa API:n käyttöön oikeuttavan tokenin.

5.1.1 Spring Boot

Spring on viitekehys, joka tarjoaa infrastruktuurin Java-ohjelmistojen kehitykseen ja sisältää lukuisia moduuleita, eli valmiita ratkaisuja erilaisiin ongelmiin ja toiminnallisuuksiin. Käyttämällä ja soveltamalla Springin moduuleita on mahdollista huomattavasti vähentää ohjelmistojen kehitykseen tarvittavaa aikaa. Spring Bootia voidaan pitää Spring-viitekehyksen laajennuksena, joka ottaa näkemyksellisen kannan Springin käyttöön tarjoamalla oletuskonfiguraatiot toiminnallisuuksille, ja näin ollen poistaa tarpeen niin sanotulle boilerplate-koodille. (Baeldung 2023.)

Kuvasta 6 on nähtävissä tässä projektissa apuna käytetyt Spring-moduulit. Data JPA on tiedon käyttökerros, eli muun muassa tietokantayhteys ja DDL- ja DML-operaatiot. Security on autentikoinnin ja auktorisoinnin viitekehys Spring-ohjelmistoille. Web on moduuli web-ohjelmistojen

kehittämiseen MVC-arkkitehtuurin mukaisesti, OAuth2-moduulit ovat token-pohjaisen autentikoinnin ominaisuuksien integraatioihin ja validation on syötteiden varmentamiseen.

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
implementation 'org.springframework.boot:spring-boot-starter-security'
implementation 'org.springframework.boot:spring-boot-starter-web'
implementation 'org.springframework.security:spring-security-oauth2-resource-server'
implementation 'org.springframework.security:spring-security-oauth2-jose'
implementation 'org.springframework.boot:spring-boot-starter-validation'
```

Kuva 6. Projektiin lisätyt riippuvuudet build.gradle-tiedostossa

Valitsin projektiin Spring Bootin, koska siitä löytyy valmiit perusmoduulit lähes kaikkia toimintoja ja ominaisuuksia varten, joita projektissa tulen tarvitsemaan. Tämä säästää aikaa perustoimintojen koodauksessa ja mahdollistaa syvemmän perehtymisen API:n tietoturvalogiikkaan. Lisäksi Spring Boot on itselleni tutuin Java-viitekehys, jota olen aiemminkin käyttänyt Java-ohjelmistojen, myös REST API:en kehityksessä.

5.1.2 Keycloak

Keycloak on avoimen lähdekoodin identiteetti ja käyttöoikeuksien hallintaratkaisu web-sovelluksille ja -palveluille. Sen tavoite on tehdä sovelluksien turvaaminen helpoksi kehittäjille valmiiden tilannekohtaisesti kustomoitavissa olevien ominaisuuksien avulla. Keycloak tarjoaa muun muassa vahvan tunnistuksen, käyttäjien hallinnan ja hienojakoisen auktorisoinnin, sekä tukee standardi protokollia, kuten OAuth 2.0, OIDC ja SAML 2.0. (Red Hat 2020.)

API:n käyttäjien tunnistaminen on ulkoistettu Keycloakille, joka toimii tässä projektissa identiteetin tarjoajana, eli OAuth 2.0 -auktorisointipalvelimena. Käytännössä käyttäjä tunnistautuu Keycloak-ohjelmiston kautta, joka luo käyttäjän tunnistetiedot ja oikeudet sisältävän JWT:n. Toteutetun API:n toiminnallisuuksien kannalta olennaisinta on Keycloakin luoma JWT ja sen sisältö. Keycloakin käyttöönotto ja konfigurointi taas eivät oleellisesti liity API:n turvaamiseen, ja ne sijoittuvat työn rajausten ulkopuolelle eikä niitä käydä tässä työssä läpi.

API:n suojattujen päätepisteiden käyttö on mahdollista vain esittämällä validi Keycloakin luoma token, joka on listätty API:lle tehtyihin pyyntöihin HTTP-otsikoksi. Kuvassa 7 on nähtävissä tässä projektissa käytetyn JWT:n datasisältöä. Projektin kannalta tärkeimpiä tietoja ovat tokenin vanhenemis- (exp) ja luontiaika (iat), tokenin liikkeellelaskija (iss), käyttäjän ID (sub), käyttäjän roolit (roles), valtuutettu osapuoli (azp) ja tokenille sallitut palautusosoitteet (allowed-origins).

Myöhemmissä alaluvuissa autentikointi (5.3.2) ja auktorisointi (5.3.4) kuvataan tarkemmin JWT:n roolia osana tunnistautumista ja pääsynhallintaa.


```
{
  "exp": 1676997813,
  "iat": 1676997513,
  "auth_time": 1676997513,
  "jti": "b397fc20-e067-4f12-a244-8e26d696d330",
  "iss": "http://localhost:8083/auth/realms/myrealm",
  "sub": "76441ce5-0728-4cb5-90ef-52b100e882bc",
  "typ": "Bearer",
  "azp": "thesis-front",
  "nonce": "42b17d39-798c-41a9-8557-f1021f9efb6b",
  "session_state": "70027e7b-cfab-450e-9574-
3e99f3675407",
  "acr": "1",
  "allowed-origins": [
    "http://localhost:3000"
  ],
  "scope": "openid",
  "sid": "70027e7b-cfab-450e-9574-3e99f3675407",
  "roles": [
    "user"
  ]
}
```


Kuva 7. Esimerkki projektissa käytetyn JWT:n tietosisällöstä

5.2 Projektista yleisesti

Toiminnallinen osuus alkoi kevyellä suunnitteluprosessilla, päätin ensin toteuttavani minimalistisen MVC-arkkitehtuurityylisen API:n mukaillen rakennetta, jota olen aiemmin käyttänyt API-projekteissa (liite 1). Minimalistinen tässä yhteydessä tarkoittaa, että API on valitun arkkitehtuurin mukainen ja täysin toimiva, mutta eri toiminnallisuuksia on vähän. Perustoiminnallisuuksien jälkeen toteuttaisin tietoperustassa kuvatut suojausmekanismit yksi kerrallaan. Lopuksi testaisin, että implementoidut suojaukset toimisivat oikein ja halutulla tavalla.

Käytännön toteutus lähti liikkeelle uuden Spring-projektin luonnilla käyttäen apuna Spring Initializr -työkalua, jolla on helppo generoida uusi Spring Boot -projekti, lisätä siihen tarvitsemansa Spring-moduulit ja tehdä valinnat projektin ohjelmointikielestä, koontiautomaatiotyökalusta ja versioista. Projekti mukailee MVC-arkkitehtuuria, jota projektissa käytetty Spring Web noudattaa. Spring Webillä toteutetussa REST API:ssa sille tehtyjen pyyntöjen kulku on korkealla tasolla seuraava: Käyttäjä tekee HTTP-pyynnön asiakasohjelmiston kautta. Pyyntö välitetään Controller-luokalle. Controller-luokka käsittelee pyynnön ja välittää sen Service-luokalle. Service-luokka toteuttaa liiketoimintalogiikan mukaiset operaatiot, ja käsittelee tarvittaessa tietokannan dataa Repository-luokan kautta, ja palauttaa lopuksi vastauksen tai dataa Controllerille. Controller käsittelee ja muokkaa vastauksen oikeaan muotoon ja palauttaa sen asiakasohjelmistolle, joka puolestaan esittää sen käyttäjälle.

Uuden projektin luonnin jälkeen toteutin minimalistiseen REST API:iin tarvittavat komponentit. Projektissa on yksi Controller, joka sisältää neljä päätepistettä eri metodeilla (GET, POST, PUT ja DELETE). Yksi Entity-luokka, joka esittää kirjaobjektia ja kolme erilaista kirjaan liittyvää DTO-luokkaa. Sisäisesti API käsittelee kirjoja Entity-muodossa ja asiakasohjelmalle tiedot palautetaan DTO-muodossa. Sisäisten ja palautettavien objektien muuntamiseen tarvitaan Mapper-luokka, joka muuntaa Entityn DTO:ksi ja toisinpäin. Yksi Service-luokka, jonka kautta kirjoja käsitellään liiketoimintalogiikan mukaisesti. Yksi Repository-luokka, jonka kautta Service-luokka käsittelee dataa tietokannassa. (Liite 1.)

API:n perustoimintojen valmistumisen jälkeen siirryin toteuttamaan API:en yleiset suojausmekanismit, eli käytönrajoituksen, autentikoinnin, lokikirjaukset, auktorisoinnin, sekä salauksen, jotka ovat Madden (2021, 18) mukaan tehokas tapa suojautua yleisimpiä uhkia vastaan. Näiden yleisten mekanismien lisäksi projektissa on toteutettu hienojakoisempia tietoturvaa parantavia ominaisuuksia OWASP:in määrittelemiä uhkia vastaan. Implementoidut toiminnallisuudet ja ominaisuudet on testattu käyttäen apuna Postman-työkalua. Suojausmekanismien käytännön toteutustavat on valittu etsimällä kutakin mekanismia vastaava ratkaisu Spring-viitekehyksestä. Apuna valittujen ratkaisujen implementoinnissa on kätetty Springin omia dokumentaatioita, ohjeita ja malliesimerkkejä.

5.3 Toteutetut suojaukset

Spring Security on tehokas ja erittäin kustomoitava autentikoinnin ja pääsynhallinnan viitekehys Spring-pohjaisille ohjelmistoille, joka suojaa myös yleisimmiltä hyökkäyksiltä. Spring Security on tässä projektissa keskeinen osa kaikkia toteutettuja suojauksia, jotka toteutetaan yleensä sarjana suodattimia, kuten aiemmin luvussa neljä on kuvailtu. Spring Securityssä suodattimet ovat osa SecurityFilterChain-suodatinketjua. Yksittäinen pyyntö kulkee järjestyksessä suodattimelta toiselle ja saavuttaa lopulta API:n toimintalogiikan, mikäli sitä ei estetä matkalla. Suodattimien käyttö ja se mitkä tai minkä tyyppiset pyynnöt kulkevat minkäkin suodattimien läpi konfiguroidaan SecurityFilterChain-luokan implementaatiossa (liite 3).

5.3.1 Käytönrajoitus

API:n käytönrajoitus on toteutettu Googlen ylläpitämän Guava-kirjaston RateLimiteria hyväksikäyttäen. Käsitteellisesti ajateltuna RateLimiter jakaa toimintalupia ylittämättä ennalta määriteltyä raja-arvoa. Tässä projektissa arvona on käytetty QPS (Queries Per Second) -arvoa. Kuvasta 8 on nähtävissä, että QPS-arvoksi on määritelty create-metodilla viisi. Tämä tarkoittaa, että keskimäärin RateLimiter jakaa viisi lupaa sekunnissa. Lupaa kysytään tryAquire-metodilla, joka palauttaa totuusarvon sen mukaisesti onko lupia vapaana, eli käytännössä sekunnin mittaisen ajanjakson aikana viisi ensimmäistä kyselyä palauttaa tosiarvon ja sen jälkeiset epätosiarvon.

```
@Service
public class RateLimitService {
    private final static RateLimiter rateLimiter = RateLimiter.create(5d);

    public static boolean returnLimit(){
        return rateLimiter.tryAcquire();
    }
```

Kuva 8. RateLimitService-luokka

Madden (2021, 66–67) toteaa, että optimaalisinta on tehdä käytönrajoitus mahdollisimman aikaisessa vaiheessa, esimerkiksi API-yhdyskäytävän tasolla, mutta lisäksi myös yksittäisen API-palvelimen tasolla. Tässä projektissa ei käytetä API-yhdyskäytävää, joten rajoitus tehdään vain API:n tasolla kustomoidun suodattimen avulla. Kuvassa 9 nähdään rajoituksen suorittavan suodattimen logiikka. Pyynnön saapuessa suodattimelle kutsutaan ensin RateLimitServicen returnLimit-metodia, joka palauttaa totuusarvon sen mukaan onko lupia vapaana (kuva 8). Pyynnön käsittely tapahtuu palautuneen arvon mukaisesti, joka tarkastetaan if-else-lauseella. Epätosiarvolla pyyntö estetään ja vastaus palautetaan heti tai tosiarvolla pyyntö lähetetään doFilter-metodilla eteenpäin seuraavalle suodatinketjun suodattimelle.

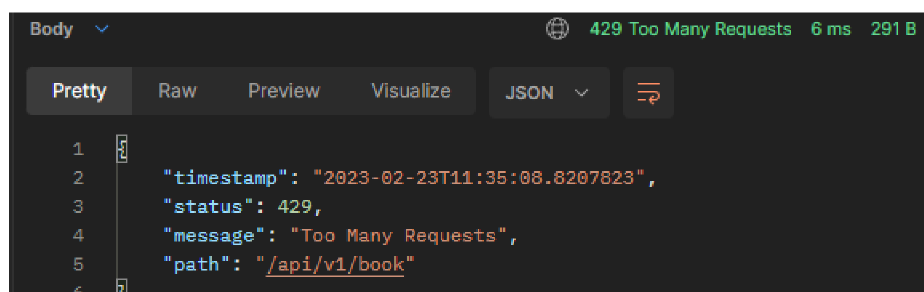
```
public class RateLimitFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain
    ) throws ServletException, IOException {
        boolean limit = RateLimitService.returnLimit();

        // If rate limit is exceeded -> respond with 429 Too many requests
        if (!limit){
            // Code omitted

        // If request is within rate limit -> continue to downstream filters
        }else{
            filterChain.doFilter(request,response);
        }
    }
}
```

Kuva 9. RateLimitFilter-luokka (lyhentämätön lähdekoodi liitteessä 2)

Suodatin ei ole käytössä ennen kuin se lisätään osaksi suodatinketjua. Lisääminen tapahtuu SecurityFilterChain-luokassa käyttämällä esimerkiksi metodia addFilterBefore, jolla suodatin voidaan lisätä haluttuun kohtaan ketjua. Rajoitin on lisätty tässä projektissa ketjun ensimmäiseksi suodattimeksi. (Liite 3.)

Suodattimen toimintaa voi testata lähettämällä API:lle määritellyn raja-arvon ylittävä määrä pyyntöjä. Kuvasta 10 nähdään API:n vastaus, kun API:lle on Postman-työkalun kautta lähetetty liian monta pyyntöä lyhyen ajan sisällä. API palauttaa HTTP-statuksen “429 Too Many Requests”, joka

on tilanteeseen sopiva ilmoitus, kuten IETF:n (Internet Engineering Task Force) RFC 9110-standardi suosittelee.



Kuva 10. API:n vastaus pyyntöjen ylittäessä sallitun määrän

5.3.2 Autentikointi

Token-pohjainen autentikointi on käytetyin tyyli API:en kohdalla, ja JWT on yleisimmin käytetty token-muoto (Levin 2019; Madden 2021, 99,191). Siksi tämä yhdistelmä on valittu myös tähän projektiin. Autentikointi API:lle tapahtuu JWT:llä Bearer Token -tyylillä, eli pyynnön HTTP-otsikkoihin lisätään avain-arvo-pari, jossa avain on Authorization ja arvo on “Bearer <token>” (<token> korvataan käyttäjän tokenilla). API vahvistaa tokenin aitouden sen liikkeelle laskeneelta auktorisointipalvelimelta, eli tässä projektissa Keycloakilta. Pääsy evätään, jos tokenia ei hyväksytä, esimerkiksi jos sitä on muutettu tai se on vanhentunut. Validi token taas oikeuttaa API:n niiden päätepisteiden käyttöön, jotka on määritelty vaatimaan tunnistautumisen.

Spring Security tukee valmiiksi useita eri autentikointitapoja, ja näitä valmiita ratkaisuja hyväksikäyttämällä autentikointi on helppo toteuttaa. Käytännössä tässä projektissa autentikoinnin toteuttaminen vaati vain autentikointikulun konfiguroimisen halutunlaiseksi sekä auktorisointipalvelimen osoitteen määrittelyn. Myös autentikointimekanismit ovat suodattimia, joten niiden määrittely tapahtuu SecurityFilterChain-luokassa.

Toteutetun API:n neljästä päätepisteestä yksi on määritelty julkiseksi, ja muita kutsuttaessa tarvitsee esittää validi token. Kuvasta 11 on nähtävissä konfiguraatiot, jotka on lisätty SecurityFilterChainiin tämän toiminnallisuuden saavuttamiseksi. Pääsyn rajoittaminen URL-perusteisesti otetaan käyttöön authorizeHttpRequests-metodilla, ja requestMatchers-metodilla määritellään rajoituksen kohteena olevan resurssin URL ja HTTP-metodi. Tässä käsittelyn kohteen osoite on /api/v1/book ja metodi on GET. Kutsu tähän resurssiin on määritelty julkiseksi permitAll-metodilla, ja tämän jälkeen anyRequest.authenticated-metodi määrittelee kaikki muut resurssit vaatimaan tunnistautumista. Metodi oauth2ResourceServer ottaa käyttöön ja konfiguroi OAuth 2.0 Resource Server -tuen token-pohjaista autentikointia varten ja jwt-metodi määrittelee käytettävän tokenin olevan

tyyppiä JWT. Koodilohkon viimeisen rivin jwtAuthenticationConverter-metodi liittyy roolipohjaiseen auktorisointiin, johon palataan myöhemmin alaluvussa auktorisointi (5.3.4).

```
// Security for HTTP requests enabled
.authorizeHttpRequests(authorize -> authorize
        // GET method for /api/v1/book endpoint is public
        .requestMatchers(HttpMethod.GET, ...patterns: "/api/v1/book").permitAll()
        // All other requests require authentication
        .anyRequest().authenticated()
)
// Configuration for JWT authentication
.oauth2ResourceServer(oauth2 -> oauth2.jwt()
        .jwtAuthenticationConverter(jwtRoleAuthenticationConverter())
)
```

Kuva 11. SecurityFilterChainissa tehdyt HTTP-pyyntöjen autentikoinnin konfiguraatiot

API:n tarvitsee varmentaa jokaisen tokenin aitous. Tätä varten tarvitsee konfiguroida jwk-set-uri-osoite, joka on kuvan 12 mukaisesti tehty application.properties-tiedostossa. Tästä osoitteesta löytyy JWT:n liikkeellelaskijan julkinen avain, jolla token voidaan varmentaa. Lisäksi on konfiguroitu issuer-uri, joka on tokenin liikkeellelaskijan osoite, josta API hyväksyy tokeneita, ja jonka tulee vastata tokenin sisältämän iss-kentän, eli liikkeellelaskijakentän arvoa. Tässä projektissa liikkeellelaskija on paikallisesti käytössä oleva Keycloak-ohjelmisto.

```
## OAuth 2.0 Resource server config
spring.security.oauth2.resourceserver.jwt.issuer-uri=\
  ${ISSUER_URL:http://localhost:8083/auth/realms/myrealm}
spring.security.oauth2.resourceserver.jwt.jwk-set-uri=\
  ${JWK_SET_URI:http://localhost:8083/auth/realms/myrealm/protocol/openid-connect/certs}
```

Kuva 12. OAuth 2.0 Resource server -konfiguraatiot application.properties-tiedostossa

Liitteessä 5 on nähtävissä kolmen suoritetun autentikointitestin tulokset. Ensimmäisessä testissä tehdään GET-pyyntö julkiseksi määriteltyyn api/v1/book-osoitteeseen ilman JWT:tä, ja odotetusti API sallii pyynnön ja palauttaa vastauksen “200 OK”, joka on standardi vastaus onnistuneelle HTTP-pyynnölle. Toisessa testissä tehdään POST-pyyntö samaan osoitteeseen ilman JWT:tä, johon API:n vastaus on “401 Unauthorized”, joka taas on standardi vastaus, kun autentikointi on epäonnistunut tai sitä ei suoritettu. Testin 2 vastaus oli myös odotettu, koska ainoastaan testin 1 mukainen pyyntö on määritelty API:ssa saataville ilman autentikointia. Kolmannessa testissä toistetaan testin 2 pyyntö, mutta lisätään siihen tunnistautuneen käyttäjän JWT, johon API:n vastaus on “201 Created”, joka on yleinen vastaus onnistuneeseen POST-pyyntöön. Testien perusteella API toimii autentikoinnin osalta odotetusti vastaten tehtyä konfigurointia (kuva 11).

5.3.3 Valvontalokin kirjaus

Käytönrajoituksen suorittavan suodattimen tavoin, myös valvontalokikirjaukset on toteutettu kustomoidulla suodattimella. Luvussa 4.3 kuvaillun parhaan käytänteen mukaisesti valvontalokikirjaus tehdään aina pyynnöstä sekä vastauksesta, pois lukien ainoastaan tilanne, jossa rajoitin estää pyynnön. Pyynnöstä kirjataan sen URI, metodi, tapahtuma-aika, sekä alkuperä. Vastauksesta kirjataan edellä mainittujen lisäksi statuskoodi, käyttöoikeuden taso, sekä pyynnön tehneen käyttäjän ID, ja mikäli pyyntö hylätään, kirjataan hylkäämisen syy. Kirjattavat tiedot mukailevat Kybertuvallisuuskeskuksen (2023) ohjeita hyvästä lokikirjauksesta.

Kuvasta 13 on nähtävissä pyyntöjen lokituksen toteutus, joka tehdään AuditLoggingFilter-suodatinluokassa, jonka koko lähdekoodi on nähtävissä liitteestä 4. Pyynnön saapuessa suodattimelle luodaan ensin StringBuilder-objekti, johon lisätään halutut tiedot pyynnöstä, getRequestURI-metodi hakee kutsutun päätepisteen osoiteen, getMethod-metodi hakee pyynnössä käytetyn HTTP-metodin ja getRemoteHost-metodi hakee pyynnön lähteen IP-osoitteen, sekä LocalDateTime.now-metodi luo aikaleiman. LogWriter.write on apuluokan metodi, joka kirjoittaa sille syötetyn datan tiedostoon. Lopuksi doFilter-metodilla pyyntö lähetetään eteenpäin suodatinketjun seuraavalle suodattimelle.

```
protected void doFilterInternal(HttpServletRequest request,HttpServletResponse response,FilterChain filterChain
) throws ServletException, IOException {
    // Log request
    StringBuilder sbr = new StringBuilder("\nAuditLoggingFilter Pre:");
    sbr.append("\n     URI: ").append(request.getRequestURI())
            .append("\n     METHOD: ").append(request.getMethod())
            .append("\n     ORIGIN: ").append(request.getRemoteHost())
            .append("\n     TIME: ").append(LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
    // Write to file
    LogWriter.write(sbr.toString());
    // Call next filter of the chain
    filterChain.doFilter(request, response);
```

Kuva 13. AuditLoggingFilter-luokassa tehty pyynnön lokikirjaus

API:n vastauksen lokikirjaus onnistuu samassa AuditLoggingFilter-luokassa, jossa pyynnön lokikirjaus tehtiin, sillä jokaisella kutsutulla suodattimella on vielä mahdollisuus suorittaa operaatioita vastaukselle ennen sen palauttamista pyynnön tehneelle asiakasohjelmalle. Kuvasta 14 on nähtävissä vastauksen lokitus. Vastauksesta kirjataan samat tiedot kuin pyynnöstä, joiden lisäksi kirjataan getStatus-metodilla HTTP-statuskoodi, getName-metodilla pyynnön tehneen käyttäjän ID, getAuthorities-metodi hakee käyttöoikeuden tason, ja lopuksi ehdollisena kirjataan autentikoinnin epäonnistumisen syy mikäli challenge-muuttuja ei ole tyhjä, eli autentikointi on epäonnistunut. LogWriter.write-metodi jälleen kirjoittaa syötetyn datan tiedostoon. Myös lokikirjauksen tekevä suodatin

on lisättävä osaksi suodatinketjua. Lisäys on tehty SecurityFilterChainissa addFilterBefore-metodilla edeltämään autentikoinnin suorittavaa suodatinta (liite 3).

```
filterChain.doFilter(request, response);
// Log response
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
String challenge = response.getHeader( name: "www-authenticate");
// Log entry base data
StringBuilder sb = new StringBuilder("\nAuditLoggingFilter Post:");
sb.append("\n     URI: ").append(request.getRequestURI())
        .append("\n     METHOD: ").append(request.getMethod())
        .append("\n     ORIGIN: ").append(request.getRemoteHost())
        .append("\n     TIME: ").append(LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME))
        .append("\n     STATUS: ").append(response.getStatus())
        .append("\n     USER: ").append(auth != null ? auth.getName() : null)
        .append("\n     AUTH: ").append(auth != null ? auth.getAuthorities() : null);
// Log entry additional data on authentication failure
if (challenge != null){
    sb.append("\n     MSG: ")
            .append(challenge.contains("Bearer error") ? challenge : "Bearer token not provided");
}
// Write to file
LogWriter.write(sb.toString());
```

Kuva 14. AuditLoggingFilter-luokassa tehty vastauksen lokikirjaus

Liitteessä 6 on nähtävissä kolmen suoritetun lokikirjaustestin tulokset. Ensimmäisessä testissä on tehty GET-metodin pyyntö osoitteeseen api/v1/book, toisessa testissä on käytetty samaa osoitetta, mutta metodilla POST ja lisäämällä validi token, kolmannessa testissä on myös sama osoite metodilla POST, mutta virheellisellä tokenilla. Jokaisen testipyynnön kohdalla odotetusti lokiin kirjautui kaksi tapahtumaa, yksi pyynnöstä ja yksi vastauksesta. Kaikkien tapahtumien tiedot kirjautuivat oikein AuditLoggingFilter-luokassa määritellyllä tavalla (kuva 13,14).

Tässä toteutuksessa lokikirjaukset tehdään kirjoittamalla data tekstimuodossa tiedostoon. Tämä ei tuotantoympäristökäytössä olisi tehokkain ratkaisu muun muassa analysoinnin kannalta. Ratkaisu on tehty ajankäytöllisistä syistä, huomioiden että lokitietojen säilöntä ja jälkikäsittely ovat tämän työn rajausten ulkopuolella. Säilöntämuodon muuttaminen ei myöskään oleellisesti vaikuta kirjausprosessin logiikkaan.

5.3.4 Auktorisointi

Pääsynhallintatyyliksi on valittu luvussa 4.4 kuvailtu RBAC, eli roolipohjainen tyyli, jossa roolit määräytyvät liiketoimintalogiikan mukaisesti. Rooleja on tässä projektissa kahta tasoa, jotka ovat ROLE_user ja ROLE_admin. Käyttäjien roolit määritellään Keycloakin hallintakonsolin kautta ja ovat osa JWT:n tietosisältöä, kuten aiemmin esitetystä kuvasta 7 on nähtävissä. API:n puolella määritellään eri resurssien vaatimat oikeustasot. Esimerkiksi jokin resurssi voidaan määritellä

sallituksi vain admin-roolin omaaville käyttäjille, API tarkastaa pyynnön saatuaan löytyykö pyynnön HTTP-otsikoista validi JWT, joka sisältää roolin admin ja estää pyynnön mikäli näin ei ole.

Spring Security tukee useita tapoja roolipohjaisen pääsynhallinnan implementointiin, ja tässä projektissa se on toteutettu metoditasolla Controller-luokassa. Kuvassa 15 resurssin poistamisen suorittava DELETE-metodin päätepiste on merkitty @PreAuthorize-notaatiolla sallituksi vain käyttäjille, joiden rooli on admin. Metoditason pääsynhallinnan käyttö vaatii @EnableMethodSecurity-notaation lisäämistä luokkaan, jossa SecurityFilterChain sijaitsee (liite 3).

```
@DeleteMapping(path = "/{book_id}")
@PreAuthorize("hasAuthority('ROLE_admin')")
public ResponseEntity<?> deleteBook(@PathVariable int book_id){
    try {
        bookService.deleteById(book_id);
        return ResponseEntity.noContent().build();
```

Kuva 15. Metoditason pääsynhallinnan asettaminen päätepisteeseen Controller-luokassa

@PreAuthorize tarkastaa pääsyoikeuden tason GrantedAuthority-objektista, jonne Spring Security väliaikaisesti tallentaa tiedot autentikoidun käyttäjän toimittamasta JWT:stä. Spring ei osaa automaattisesti muuntaa JWT:n sisältämiä rooleja GrantedAuthorityksi, joten tarvitaan kustomoitu metodi, joka syötetään parametrina aiemmin autentikointiosiossa mainitulle ja kuvassa 11 nähtävälle jwtRoleAuthenticationConverter-metodille. Toteutettu metodi käyttää Spring Securityn oletusasetuksien muunninluokkaa, joka konfiguroidaan tilanteeseen sopivaksi (kuva 16). Metodilla setAuthoritiesClaimName asetetaan muunnin etsimään JWT:stä avainta roles, ja setAuthorityPrefix-metodilla asetetaan roles:in sisältämille rooleille etuliite “ROLE_”. Nyt autentikoinnin yhteydessä, jos esimerkiksi JWT:n roles sisältää roolin admin, lisätään se GrantedAuthority-objektiin muodossa ROLE_admin, joka oikeuttaisi muun muassa kuvan 15 metodin kutsumiseen.

```
public JwtAuthenticationConverter jwtRoleAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();
    // Add authorities from roles claim
    grantedAuthoritiesConverter.setAuthoritiesClaimName("roles");
    // Add ROLE_ prefix for authorities
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");
    JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter);
    return jwtAuthenticationConverter;
}
```

Kuva 16. SecurityConfig-luokan jwtRoleAuthenticationConverter-metodi

Liitteessä 7 on nähtävissä kahden suoritetun auktorisointitestin tulokset. Ensimmäisessä testissä ROLE_user tason käyttöoikeuden omaava käyttäjä yrittää kutsua ROLE_admin oikeuden vaativaa

resurssia, eli ilman riittävää käyttöoikeutta. API:n vastaus on “403 Forbidden” indikoiden, että käyttäjällä ei ole oikeutta kutsutun resurssin käyttöön. Lokikirjauksesta on myös nähtävissä viesti “Bearer error=’insufficient_scope’”, eli tokenin käyttöoikeus ei ole tarpeeksi laaja resurssin käyttöön. Toisessa testissä ROLE_admin tason käyttöoikeuden omaava käyttäjä kutsuu samaa resurssia ja saa vastaukseksi “204 No Content”, eli pyyntö onnistuneesti suoritettu, joka on vastaavasti todettavissa lokikirjauksesta.

5.3.5 Salaus

API on konfiguroitu sallimaan vain salattu HTTPS-liikenne. Tämä tapahtuu application.properties-tiedostossa (kuva 17). SSL (Secure Sockets Layer) -tuen lisäämiseksi Spring Boot ohjelmistoon asetetaan server.ssl.enabled-parametrin arvoksi tosi ja server.ssl.protocol-parametrin arvolla määritellään käytetty protokolla, joka on TLS. Salattua yhteyttä varten tarvitaan myös TLS-sertifikaatti. Projektissa on käytetty itse allekirjoitettua sertifikaattia, joka on luotu JKS:llä (Java KeyStore), kuvan server.ssl.key-store-alkuiset konfiguraatiot osoittavat mistä sertifikaatti löytyy, ja miten siihen pääse käsiksi. Kun SSL konfiguroidaan käyttöön tällä tavalla, API ei enää halutusti tue pelkkää salaamatonta HTTP-liikennettä, jonka käyttö on OWASP (2019, 20) mukaan turvallisuusriski.

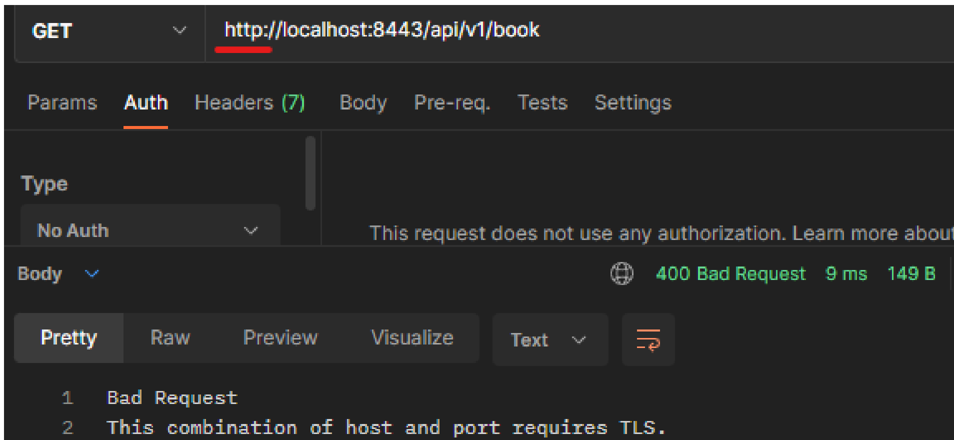
```
server.ssl.enabled=true
server.ssl.protocol=TLS
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:kstore.p12
server.ssl.key-store-password=thapi123
```

Kuva 17. HTTPS-liikenteen käyttöä varten tehdyt konfiguraatiot application.properties-tiedostossa

SSL:n konfiguroinnin jälkeen sen toimivuus voidaan varmentaa API:n käynnistyessä lokitiedoista, josta on nähtävissä, että API:n sulautettu web-palvelin on käynnistetty HTTPS:lle (kuva 18). Toiminta voidaan vielä varmistaa testillä, jossa API:lle lähetetään pyyntö HTTP:tä käyttäen (kuva 19). Tähän API:n vastaus on “400 Bad Request”, eli pyyntöä ei voitu toteuttaa käyttäjän tekemän virheen vuoksi, sekä ohjeistus käyttämään TLS-yhteyttä.

```
: Will secure any request with [com.thesis.api.filters.RateLimitFilter@142f9dc
: LiveReload server is running on port 35729
: Tomcat started on port(s): 8443 (https) with context path ''
: Started ApiApplication in 5.945 seconds (process running for 6.451)
```

Kuva 18. Lokikirjaus API-palvelimen käyttämästä HTTPS-yhteydestä



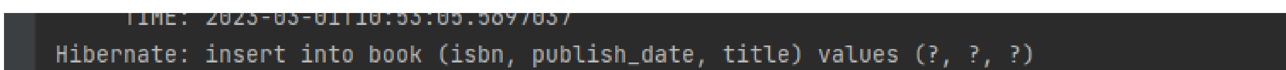
Kuva 19. API:n vastaus, kun pyyntöä ei ole tehty salatun yhteyden kautta

5.3.6 Muut suojaukset

Yleisten suojausmekanismien lisäksi on toteutettu syötteiden ja vastausten validointi, CORS:in käyttöönotto ja konfigurointi, sekä turvallisuutta parantavien HTTP-otsikoiden lisäys API:n vastauksiin.

Syötteiden ja vastausten validointi

Spring JPA:n oletusimplementaatio on toteutettu Hibernatella, joka käyttää parametrisoituja tietokantakyselyitä, eli SQL-injektiosuojaus on projektissa oletuksena. Parametrisoitujen kyselyiden käyttö on myös todennettavissa API:n lokitiedoista, kun API:lle tehdään tietokantaan tietoa lisäävä pyyntö, kuten kuvasta 20 on nähtävissä. Lisäksi syötteet validoidaan määrittelemällä syötteille sallitut arvot. Validoinnit on toteutettu OWASP (2019, 23) ja Madden (2021, 48–52) kuvailemia parhaita käytänteitä mukaillen.



Kuva 20. Lokikirjaus Hibernaten luomasta parametrisoidusta kyselystä

Syötteiden sallitut arvot on määritelty notaatioilla luokassa, jonka objekteja API käyttää sisäisesti. @Pattern-notaation regexp-parametrilla määritellään säännöllistä lauseketta käyttämällä sallitut syötteet ja niiden muoto. Esimerkiksi kuvan 21 koodilohkon kolmannella rivillä sallituiksi merkeiksi on määritelty isot ja pienet kirjaimet väliltä a ja z, numerot nollan ja yhdeksän väliltä, sekä erikoismerkit väliviiva, heittomerkki, kaksoispiste, piste ja pilkku, jotka voivat esiintyä missä tahansa järjestyksessä. @NotBlank-notaatio määrittelee, ettei kenttä voi olla tyhjä ja @PastOrPresent-notaatio määrittää sallitun aika- tai päivämäärä arvon olevan menneisyydessä tai nykyhetkessä.

Notaatioiden message-parametreilla on myös määritelty viestit, jotka lisätään virheilmoitukseen, mikäli annettu arvo ei ole sallittu. (Kuva 21.)

```
@Column(nullable = false, length = 150)
@Size(max = 150, message = "Title maximum length is 150 characters")
@Pattern(regexp = "^[a-zA-Z0-9-':., ]+$", message = "Title contains forbidden characters")
@NotBlank(message = "Title cannot be null")
private String title;
@Column(nullable = false, length = 17)
@Pattern(regexp = "[0-9-]{17}", message = "Invalid ISBN format")
@NotBlank(message = "ISBN cannot be null")
private String isbn;
@Column(nullable = false)
@NotNull(message = "Publish date cannot be null")
@PastOrPresent(message = "Publish date cannot be in the future")
private Date publishDate;
```

Kuva 21. Book-luokassa käytetyt syötteiden validoinnin notaatiot

Controller-luokassa on lisäksi määritelty missä muodossa päätepiste ottaa dataa vastaan. Kuvassa 22 @PostMapping-notaation consumes-parametrilla ainoaksi hyväksytyksi mediatyypiksi on määritelty JSON. @RequestBody-notaatio puolestaan osoittaa, että API:lle tehdyn pyynnön body-osiossa tulisi toimittaa tiedot NewBookDTO-luokan objektin luomiseksi. Spring muuntaa automaattisesti pyynnön mukana tulleen JSON-muotoisen datan @RequestBodylla määritellyn luokan objektiksi, joka on tässä NewBookDTO, kunhan JSON-avaimet vastaavat määritellyn luokan kenttien nimiä.

```
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
@PreAuthorize("hasAuthority('ROLE_user')")
public ResponseEntity<BookDTO> addBook(@RequestBody NewBookDTO newBookDTO){
    try {
        Book book = bookService.add(bookMapper.newBookDTOToBook(newBookDTO));
        String baseUrl = ServletUriComponentsBuilder.fromCurrentRequestUri().toUriString();
        URI uri = URI.create(baseUrl + "/" + book.getId());
        return ResponseEntity.created(uri).body(bookMapper.bookToBookDTO(book));
    }catch (ConstraintViolationException ex) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, validationViolations(ex));
    }
```

Kuva 22. Controller-luokassa sijaitseva POST-metodin päätepiste

API:n vastauksista on varmennettu, että ne eivät paljasta tietoa, joka saattaisi vaarantaa API:n tietoturvan. Spring tuottaa automaattisesti virheilmoituksia eri tilanteissa ja muun muassa validointivirheiden seurauksena pyynnön tekijälle palautettava oletusviesti paljastaa liikaa tietoa API:n toimintalogiikasta, kuten sisäisistä objekteista. Virheitä mahdollisesti aiheuttavat metodit kutsutaan try-catch-lausunnon sisältä, jotta validointivirheen tapahtuessa oletusviesti voidaan korvata kustomoidulla viestillä.

Esimerkiksi kuvan 22 bookService.add-metodi voi aiheuttaa ConstraintViolationException-validointivirheen, jonka sisältämät tarkat tiedot virheestä välitettäisiin oletusarvoisesti vastauksessa pyynnön tekijälle. Tämän sijasta ResponseStatusException-luokkaa käyttämällä pyynnön tekijälle palautetaan kustomoidun validationViolations-metodin avulla vain virheviestit, jotka aiemmin määriteltiin Book-luokassa notaatioiden message-parametreissa sekä “400 Bad Request”-statuskoodi. Kuvasta 23 on nähtävissä validationViolations-metodin toiminta. Metodi ottaa parametrina vastaan validointivirheistä aiheutuvan ConstraintViolationException-luokan objektin, josta se poimii vain virheviestit ja palauttaa ne String-objektina.

```
private String validationViolations(ConstraintViolationException e){
    Set<ConstraintViolation<?>> violations = e.getConstraintViolations();
    StringBuilder sb = new StringBuilder();
    violations.forEach(v-> sb.append(v.getMessage() + ", "));
    sb.delete(sb.length()-2,sb.length());
    return sb.toString();
```

Kuva 23. Controller-luokan validoinnin virheilmoitusten rajaamiseen käytetty metodi

API:n application.properties-tiedostossa on myös konfiguroitu, ettei API ikinä virheen tapahtuessa palauta vastauksessa pinojäljitystä (stacktrace), eli listausta virheeseen johtaneista metodikutsuista, pyynnön tekijälle (kuva 24).

```
27  ## Exclude stacktrace from responses
28  server.error.include-stacktrace=${INCLUDE_STACKTRACE:never}
```

Kuva 24. Stacktrace-konfiguraatio application.properties-tiedostossa

Lisäksi API:n yhdessä vastauksessa palauttamaa tietomäärää on rajoitettu sivuttamalla, kuten OWASP (2019, 14) ohjeistaa tekemään, jottei yksittäinen pyyntö voi aiheutaa ylisuurta kuormaa tietokannalle. Pyynnön tekijälle palautetaan yksi määrämittainen “sivu” dataa, eikä kaikkea kerralla. Esimerkiksi Controller-luokassa API:n julkinen päätepiste on konfiguroitu @RequestParam-notaatiolla ottamaan vastaan parametrina sivunumeron, jolla käyttäjä määrittelee minkä sivun haluaa hakea. Mikäli käyttäjä ei määrittele sivunumeroa, API palauttaa oletuksena ensimmäisen sivun sisältämän datan, joka on määritelty asettamalla defaultValue-parametriksi arvo nolla. (Kuva 25.)

```
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<BookPageDTO> findAllBooks(@RequestParam(defaultValue = "0") int page){
    Page<Book> bookPage = bookService.findAllByPage(page);
    return ResponseEntity.ok(bookMapper.bookPageToBookPageDTO(bookPage));
```

Kuva 25. Controller-luokassa sijaitseva GET-metodin päätepiste

Liitteessä 8 on nähtävissä kolmen suoritetun validointitestin tulokset. Ensimmäisessä testissä käyttäjän syöte ei läpäise validointia, jolloin API:n vastaus on “400 Bad Request”, sekä listaus ilmenneistä virheistä. Toisessa testissä pyynnön mediatyyppi on väärä, johon vastaus on “415 Unsupported Media Type”, sekä viesti: API ei tue pyynnössä käytettyä mediamuotoa. Kolmas testi on pyyntö resurssille, joka sivuttaa vastaukset. Vastauksesta käy ilmi, mikä sivu palautettiin, montako elementtiä sivu sisältää, elementtien sisältämä data, tietokannassa sijaitsevien sivujen ja elementtien kokonaismäärä, sekä odotetusti status “200 OK”. Testausten perusteella API toimii halutulla tavalla, eikä virheiden ilmetessä palauta turvallisuutta vaarantavaa tietoa sekä sivuttaa palautetun datan.

CORS

CORS:n käyttö rajoittaa pyyntöjen alkuperää, joihin API voi palauttaa dataa. Projektissa CORS:n käyttöönotto on helppo tehdä, se konfiguroidaan käyttöön API:n SecurityFilterChainissa cors.and-metodilla (kuva 26). Lisäksi halutut pyyntöjen sallitut alkuperät on konfiguroitu Controller-luokassa @CrossOrigin-notaation origins-parametrilla (kuva 27). Projektin API on konfiguroitu sallimaan pyynnöt vain osoitteista localhost:3000 ja web.postman.co, joiden kautta API:a on testattu.

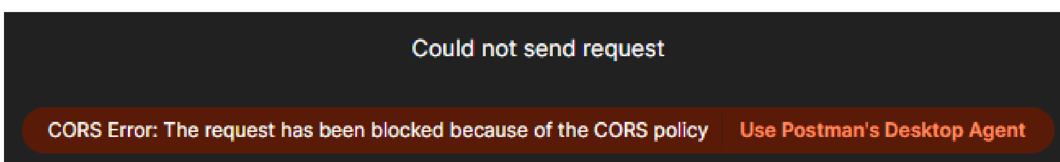
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http    // CORS Enabled
            .cors().and()
```

Kuva 26. SecurityFilterChainissa tehty CORS-konfiguraatio

```
@RestController
@CrossOrigin(origins = {"http://localhost:3000","https://web.postman.co"})
@RequestMapping(path = "api/v1/book")
public class BookController {
```

Kuva 27. Controller-luokassa tehty CORS-konfiguraatio

CORS:n toimivuutta on testattu poistamalla Postmanin osoite API:n sallituista alkuperistä, jonka jälkeen sen kautta on tehty GET-pyyntö API:n api/v1/book-osoitteeseen. Kuvasta 28 on nähtävissä Postmanin esittämä CORS-virheilmoitus, kun API ei ole sallinut sen tehdä pyyntöä.



Kuva 28. CORS-virheilmoitus

HTTP-otsikot

Spring Security lisää vastauksiin oletusarvoisesti joukon tietoturvaa parantavia HTTP-otsikoita (liite 9), jotka ovat osa OWASP:in (2023b) Secure Headers -projektin suosituksia. Oletusarvoiset tietoturvaotsikot suojaavat muun muassa clickjacking-hyökkäyksiltä, tietoturvan kannalta heikkojen protokollien käytöltä ja estävät vastausten tallentamisen välimuistiin. Näiden lisäksi projektiin on lisätty CSP (Content-Security-Policy) -otsikko, joka rajoittaa hyökkääjän mahdollisia toimia, mikäli hän onnistuu hyödyntämään jotain XSS-haavoittuvuutta (kuva 29). Projektissa käytetty otsikko on Madden (2021, 58) suosittelema minimaalinen CSP-otsikko API käyttöön, jossa policyDirectives-metodilla on määritelty kolme ohjetta, jotka estävät skriptien lataamista ja suorittamista.

```
// Add CSP header
.headers(headers -> headers
        .contentSecurityPolicy(csp -> csp
                .policyDirectives("default-src 'none'; frame-ancestors 'none'; sandbox;")
        ))
```

Kuva 29. SecurityFilterChainissa tehty CSP-otsikon konfiguraatio

6 Pohdinta

Opinnäytetyön tavoitteena oli selvittää API:en tietoturvallisuuteen vaikuttavia tekijöitä ja parhaita käytänteitä, sekä näitä tietoja hyödyntäen toteuttaa tietoturvallinen API käytännössä. Tietoperustassa käytiin ensin läpi hieman API:en historiaa, josta jatkettiin REST API:en perusteisiin ja yleiseen toimintaan. Seuraavaksi käsiteltiin API:en tietoturvallisuuspuolta periaatteiden ja suunnittelun tasolla, jonka jälkeen käytiin läpi API:en tietoturvaan vaikuttavia trendejä ja lopuksi uhkien tunnistamista.

Lisäksi tarkasteltiin kirjoitushetkellä ohjelmointirajapintoihin kohdistuvia kriittisimpiä uhkia perehtymällä OWASP:in TOP 10 -listaukseen, jossa on kuvattu uhkia, niihin johtavia tekijöitä, hyökkäysten vaikutuksia sekä torjuntakeinoja. Lopuksi perehdyttiin API:en yleisimpiin suojausmekanismeihin, jotka ovat tehokas tapa suojautua useimmilta yleisimmiltä hyökkäyksiltä.

Toiminnallisessa osuudessa kuvattiin API:n toteutusta projektin kulun, käytettyjen teknologioiden, implementoitujen suojausratkaisujen sekä suoritettujen testien osalta. Käytännön toteutus havainnollistaa, miten teoriassa kuvatut tietoturvaan vaikuttavat tekijät ja mekanismit on mahdollista toteuttaa, ja miten ne toimivat yhteen.

Projektin onnistuminen

Mielestäni työlle asetetut tavoitteet saavutettiin hyvin. Tietoperustaa laatiessa sain mielestäni kerättyä asiantuntijoiden konsensuksen mukaiset tietoturvasuositukset, ja samalla opin miten tietoturvaa kannattaa lähteä toteuttamaan, kuten myös mistä etsiä relevanttia tietoa mahdollisista uhista, käytännön toteutuksista, huomioitavista asioista sekä erilaisista ratkaisuista.

Toiminnallisessa osuudessa sain vietyä käytäntöön teoriaosuuden opit ja lopputuloksena oli tietoturvallinen API, jossa on implementoitu kaikki yleisimmät ja tehokkaimmat API:en suojausmekanismit, eikä tietoperustaan peilaten huomattavia tietoturvapuutteita jäänyt. Toteutuksen aikana opin paljon uutta käyttämästäni Spring-viitekehyksestä, muun muassa ymmärtämään ja soveltamaan paremmin jo aiemmin jossain määrin tuntemiani osia, kuten Spring Securitya. Löysin myös itselleni entuudestaan tuntemattomia aliprojekteja, joita jatkossa tai jopa tässäkin työssä olisi voinut hyödyntää esimerkiksi lokitukseen tai käytön rajoitukseen, mikäli aikataulu olisi antanut periksi niihin perehtymiseen.

API ja sen toiminnallisuudet toteutettiin lähestulkoon kokonaisuudessaan Spring Bootilla, käyttäen lisäksi muita Spring-aliprojekteja. Tämä oli kohtuullisen helppo tapa toteuttaa kaikki halutut toiminnallisuudet, sillä useat valmiit ominaisuudet sopivat projektiin suoraan oletustoteutuksillaan tai pienillä konfiguraatioilla. Yksi mieleenpainuvimmista oppimiskokemuksista olikin huomata, ettei

ohjelmiston turvallisuuden parantaminen välttämättä vaadi paljoa työtä, vaikka tietoturvallisuus onkin laaja aihe ja erilaisia uhkia, riskejä, sekä hyökkäyksiä olemassa paljon.

Vaikka projekti onnistui hyvin, parannettavaakin jäi, eikä toiminnallisessa osuudessa kaikkia parhaita käytänteitä saatu toteutettua alkuperäisen suunnitelmani mukaisesti. Lähtötilanteen näkemykseni sekä suunnitelmani oli, että kaikki suojaukset voi toteuttaa itse API:ssa. Tietoperustaa laatiessa kuitenkin selvisi, että käytännössä osa suojausmekanismeista on parempi toteuttaa API:n ulkopuolella. Esimerkiksi käytön rajoitus kannattaa tehdä yleisesti ottaen API-yhdyskäytävällä, joka on erillinen ohjelmisto, jonka kautta pyynnöt kulkevat yksittäisille API-palvelimille.

Lisäksi en saanut toteutettua kaikkia suojauksia täysin haluamallani ja parhaalla mahdollisella tavalla käyttämällä valmiita Spring-komponentteja. Esimerkiksi lokikirjaukset eivät toimi täysin yhteen autentikoinnin suodattimien kanssa tilanteissa, joissa autentikointi epäonnistuu. Tämä toiminnallisuus olisi vaatinut laajempaa autentikointikulussa käytettyjen komponenttien kustomointia. Aikataulussa pysymistä silmällä pitäen en kuitenkaan lähtenyt sitä toteuttamaan, ja pitäydyin myös alkuperäisessä suunnitelmassani toteuttamalla kaikki suojaukset itse API:ssa.

Tietoturvatietojen ja -taitojen lisäksi kehitettävää on myös muiden ohjelmistokehityksen osa-alueiden kohdalla. Etenkin suunnitteluprosessissa on itselläni vielä parannettavaa, sillä huomasin jälleen joutuvani tekemään jälkikäteen muutoksia, joilta olisi voinut välttyä perusteellisemmalla suunnittelulla. Todennäköisesti vain siirtämällä suunnitelmat päästä paperille tämänkin kaltaisissa suhteellisen pienissä projekteissa pääsisi jo parempaan tulokseen. Sen lisäksi arvioisin, että olisi järkevää käyttää enemmän aikaa muun muassa syy-seuraussuhteiden, tavoitteiden ja vaihtoehtoisten ratkaisujen pohtimiseen.

Projektin suurimmaksi riskiksi olin arvioinut aikataulun pitävyyden, johon mahdollisesti vaikuttaisivat rajaus, oma osaaminen, sekä itselleni jokseenkin tyypillinen liian optimistinen arviointi vaadittavan työn määrästä. Riskit eivät tällä kertaa realisoituneet. Työnmäärä vastasi melko lähelle arvioitua. Jotkin asiat vaativat melko suurta määrää dokumentaatioiden ja ohjeiden lukua, mutta ratkesivat lopulta, joten osaamisen kanssa ei ollut ongelmia. Työn rajaus olisi mahdollisesti voinut olla tiukempi, sillä työn aihe käsittää suhteellisen laajan määrän asioita, joista useita oli mahdollista käsitellä vain pintapuolisesti tämän kokoluokan työssä.

Opinnäytetyödokumentin laatimisessa ja kirjoittamisessa hieman yllättäen yksi haastavimmaksi kokemani asia oli käännökset. Useille ohjelmointitermeille tai -käsitteille oli mielestäni vaikea löytää hyvä käännös. Käännökset jossain määrin mielestäni myös turhaan monimutkaistavat asioita, sillä ohjelmistokehityksen pääkieli on yleisesti ottaen englanti. Tätä jälkikäteen pohtiessani olenkin sitä

mieltä, että työ olisi ollut järkevää toteuttaa englanniksi myös siksi, että työn saavutettavuus paranisi, ja siitä olisi voinut olla enemmän hyötyä uralle.

Jatkokehitystä projektille en aio tehdä, sillä se on jo ajanut asiansa käytännössä oppimisen välineenä, eikä se tule todelliseen käyttöön. Jatkokehitys- tai muutoskohteita kuitenkin löytyisi, muun muassa lisäisin aiemmin mainitun API-yhdyskäytävän, sekä korjaisin lokikirjausten ja autentikoinnin yhteentoimivuutta. Lisäksi käyttäjän syötteiden validoinnin tekisin aiemmin päätepisteiden ensimmäisenä toimintona. Nyt validointi tehdään juuri ennen tietokannan käsittelyoperaatioita, ja vaikka se ei tässä projektissa aiheuta riskejä, on virheet yleisesti ottaen parasta ottaa kiinni mahdollisimman aikaisin. Selvittäisin myös spring-boot-actuator-moduulin käyttöä, joka sisältää automaattisia lisäominaisuuksia, kuten auditoinnin, sekä tietojen keräyksen sovelluksen terveydestä ja toiminnasta erilaisten mittareiden muodossa. Actuator-moduulista olisi varmasti hyötyä ainakin tuotantoympäristökäytössä oleville ohjelmistoille.

Työn teoriaosuus on mielestäni hyvin yleisesti hyödynnettävissä, sillä se käsittelee API:en tietoturvaa korkeammalla tasolla ja antaa suuntaa siitä, miten tietoturvaa voi lähestyä suunnittelun, periaatteiden ja riskien tunnistamisen kautta. Tulevaisuuden käytettävyyttä kuitenkin hieman rajoittaa se, että työssä ovat oleellisessa osassa kirjoitushetken kriittisimmät tietoturvariskit, jotka IT-alalle ominaisella tavalla saattavat muuttua nopeastikin. Toiminnallisen osuuden hyödynnettävyys on hieman suppeampi, sillä se esittelee vain yhden tavan toteuttaa tietoturvallinen API ja nimenomaan Spring-viitekehyksen puitteissa. Implementoidut suojaukset on kuitenkin kuvattu tarkasti, joten siitä voi olla hyötyä niille lukijoille, jotka käyttävät tai aikovat käyttää samoja teknologioita kuin projektissa on käytetty.

Kaiken kaikkiaan olen tyytyväinen opinnäytetyön lopputulokseen. Olen sen aikana oppinut huomattavan määrän tietoturvallisuudesta kokonaisuutena ja erityisesti API:en suojaamisesta. Lähtötilanteeseen verrattuna tietoturvallisuus näyttäytyy itselleni huomattavasti selkeämpänä kokonaisuutena, vaikkakin paljon on vielä opittavaa. Lisäksi Java-ohjelmointitaitoni ovat edelleen parantuneet. Näkisin, että kaikesta projektin aikana opitusta on varmasti hyötyä itselleni jatkossa.

Lähteet

Aumasson, J-P. 2018. Serious Cryptography. 8. painos. No Starch Press. San Francisco. E-kirja. Luettu: 1.2.2023.

AWS. 2023. Types of access control. Luettavissa: https://docs.aws.amazon.com/prescriptive-guidance/latest/saas-multitenant-api-access-authorization/access-control-types.html. Luettu: 1.2.2023.

Baeldung. 2022. A Comparison Between Spring and Spring Boot. Luettavissa: https://www.baeldung.com/spring-vs-spring-boot. Luettu: 21.2.2023.

CIS. 2023. Election Security Spotlight – Defense in Depth (DiD). Luettavissa: https://www.cisecurity.org/insights/spotlight/cybersecurity-spotlight-defense-in-depth-did. Luettu: 25.1.2023.

Curity. 2023. JWT Security Best Practices. Luettavissa: https://curity.io/resources/learn/jwt-best-practices/. Luettu: 1.2.2023.

DataDome. 2020. What is API Rate Limiting and How to Implement It. Luettavissa: https://datadome.co/bot-management-protection/what-is-api-rate-limiting/. Luettu: 2.2.2023.

Deogun, D., Johnsson, D. B. & Sawano, D. 2019. Secure by Design. Manning Publications. New York.

EDUCBA. 2023. Authentication methods. Luettavissa: https://www.educba.com/authentication-methods/. Luettu: 31.1.2023.

Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Luettavissa: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf. Luettu: 19.1.2023.

Geekflare. 2022. Why and How to Secure API Endpoint? Luettavissa: https://geekflare.com/securing-api-endpoint/. Luettu: 16.1.2023.

Hawkins, M. 2020. The History and Rise of APIs. Luettavissa: https://www.forbes.com/sites/forbestechcouncil/2020/06/23/the-history-and-rise-of-apis/?sh=8128d4845c28. Luettu: 19.1.2023.

Hussain, F., Hussain, R., Noye, B. & Sharieh, S. 2020. Enterprise API Security and GDPR Compliance: Design and Implementation Perspective. IT Professional, 22, 5, s. 81-89.

IBM. 2023. What is a DDoS attack? Luettavissa: https://www.ibm.com/topics/ddos. Luettu: 31.1.2023.

IETF. 2022. HTTP Semantics. Luettavissa: https://datatracker.ietf.org/doc/html/rfc9110. Luettu: 7.3.2023.

Jagger, E. 2021. What's the Difference Between JSON and XML? Luettavissa: https://www.abstractapi.com/guides/json-vs-xml. Luettu: 20.1.2023.

json.org s.a. Introducing JSON. Luettavissa: https://www.json.org/json-en.html. Luettu: 19.3.2023.

Lane, K. 2019. Intro to APIs: History of APIs. Luettavissa: https://blog.postman.com/intro-to-apis-history-of-apis/. Luettu: 17.1.2023.

Lane, K. 2022. The API-First Transformation. Postman. San Francisco.

Levin, G. 2019. Four Most Used REST API Authentication Methods. Luettavissa: https://dzone.com/articles/four-most-used-rest-api-authentication-methods. Luettu: 31.1.2023.

Kyberturvallisuuskeskus. 2023. Näin keräät ja käytät lokitietoja. Luettavissa: https://www.kyberturvallisuuskeskus.fi/fi/ajankohtaista/ohjeet-ja-oppaat/nain-keraat-ja-kaytat-lokitietoja. Luettu: 25.1.2023.

Noroff. 2023. Modern web security. Luettavissa: https://noroff-accelerate.gitlab.io/java/course-notes/_rework/module4/02_WebSecurity/. Luettu: 31.1.2023.

Madden, N. 2021. API Security in Action. Manning Publications. New York.

MDN s.a. Cross-Origin Resource Sharing (CORS). Luettavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Luettu: 19.3.2023.

Microsoft. 2022. RESTful web API design. Luettavissa: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design. Luettu: 13.1.2023.

Microsoft. 2023. What is access control? Luettavissa: https://www.microsoft.com/en-us/security/business/security-101/what-is-access-control. Luettu: 2.2.2023.

OAuth s.a. OAuth 2.0. Luettavissa: https://oauth.net/2/. Luettu: 18.1.2023.

Okta s.a. Introduction to JSON Web Tokens. Luettavissa: https://jwt.io/introduction. Luettu: 30.1.2023.

OpenID s.a. What is OpenID? Luettavissa: https://openid.net/connect/. Luettu: 18.1.2023.

OWASP. 2019. API Security TOP 10 2019. Luettavissa: https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf. Luettu: 13.1.2023.

OWASP. 2023a. Threat Modeling Process. Luettavissa: https://owasp.org/www-community/Threat_Modeling_Process. Luettu: 7.2.2023.

OWASP. 2023b. OWASP Secure Headers Project. Luettavissa: https://owasp.org/www-project-secure-headers/. Luettu: 2.3.2023.

Postman. 2020. What Is a REST API? Examples, Uses, and Challenges. Luettavissa: https://blog.postman.com/rest-api-examples/. Luettu: 19.1.2023.

Red Hat. 2019. API security. Luettavissa: https://www.redhat.com/en/topics/security/api-security. Luettu: 17.1.2023.

Red Hat. 2020. Getting Started with The Keycloak Single Sign-On Operator. Luettavissa: https://www.redhat.com/en/blog/getting-started-keycloak-single-sign-operator. Luettu: 21.2.2023.

Salt. 2022. API Security Trends. Luettavissa: https://salt.security/api-security-trends. Luettu: 7.2.2023.

Sandoval, K. 2018. 3 Common Methods of API Authentication Explained. Luettavissa: https://nordicapis.com/3-common-methods-api-authentication-explained/. Luettu: 31.1.2023.

Santos, W. 2017. Which API Types and Architectural Styles are Most Used? Luettavissa: https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26. Luettu: 19.1.2023.

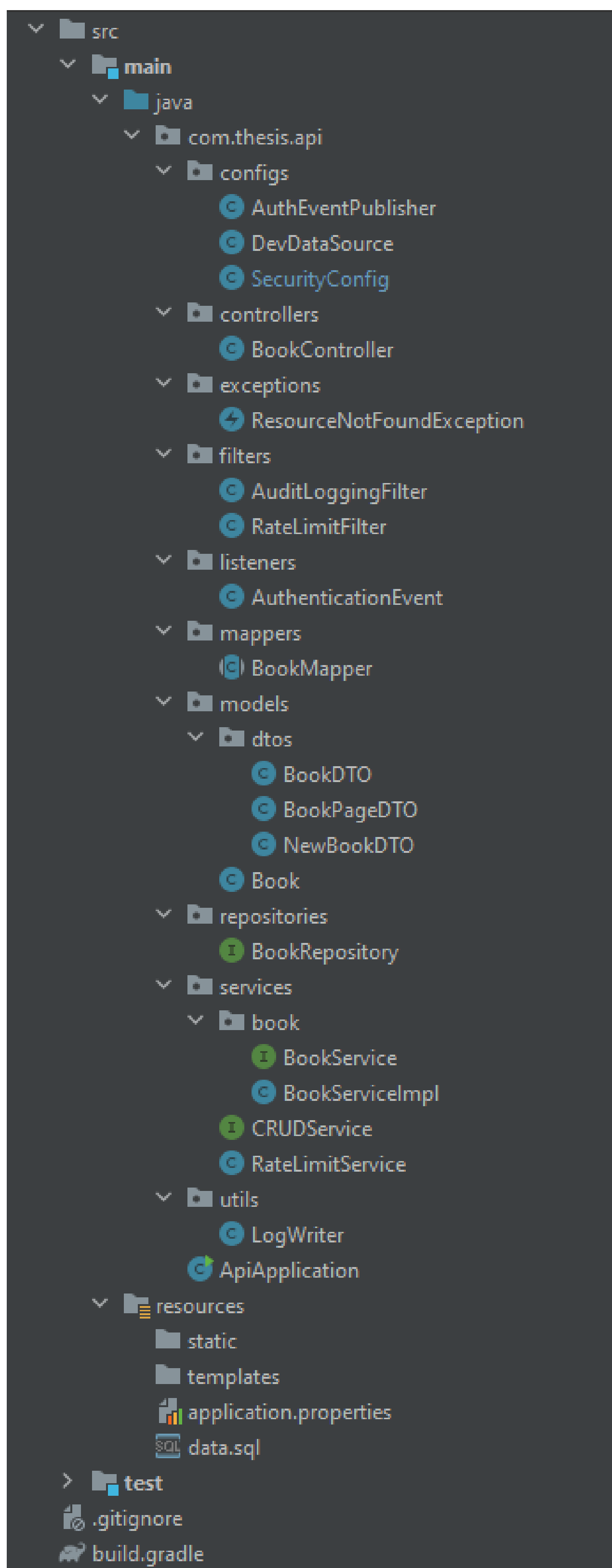
Siriwardena, P. 2020. Advanced API Security: Oauth 2.0 and Beyond. Apress. New York.

Spring s.a. Why Spring? Luettavissa: https://spring.io/why-spring. Luettu: 18.3.2023.

WebscrapingAPI. 2021. The 5 Most Popular API Styles and What Sets them Apart. Luettavissa: https://www.webscrapingapi.com/the-5-most-popular-api-styles-and-what-sets-them-apart. Luettu: 19.1.2023.

Liitteet

Liite 1. Projektin rakenne



Liite 2. RateLimitFilter-luokka

```java
public class RateLimitFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain
    ) throws ServletException, IOException {
        boolean limit = RateLimitService.returnLimit();

        // If rate limit is exceeded respond with 429 Too many requests
        if (!limit){
            LocalDateTime time = LocalDateTime.now();

            StringBuilder sb = new StringBuilder();
            sb.append("{" +
                    "\"timestamp\": \""+ time.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME) + "\"," +
                    "\"status\": 429," +
                    "\"message\": \"Too Many Requests\"," +
                    "\"path\": \"" + request.getRequestURI() + "\"" +
                    "}");

            response.resetBuffer();
            response.setStatus(429);
            response.setHeader("Content-Type", "application/json");
            response.getOutputStream().print(sb.toString());
            response.flushBuffer();

        // If request is within rate limit continue to downstream filters
        }else{
            filterChain.doFilter(request,response);
        }
    }
}
```

Liite 3. SecurityConfig-luokka

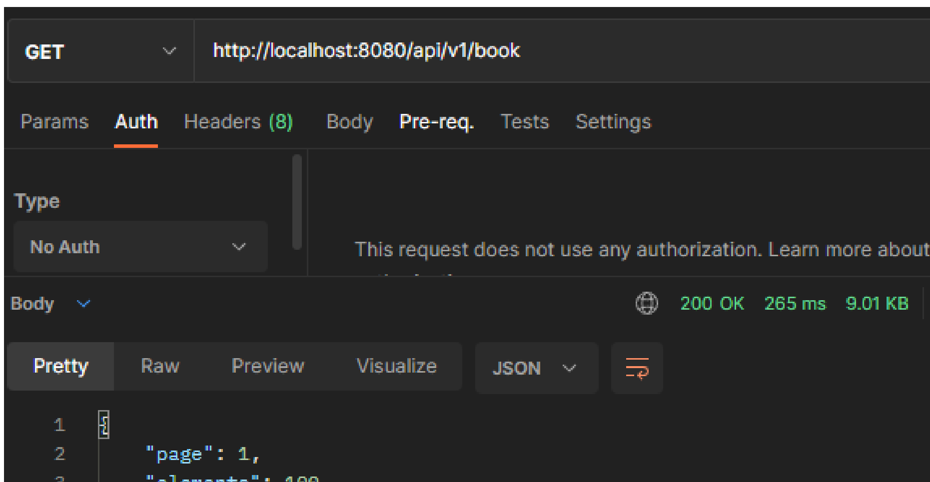
```java
@Configuration
@Profile("dev")
@EnableWebSecurity(debug = false) // Debugging off for production environment
@EnableMethodSecurity
public class SecurityConfig{
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http    // CORS Enabled
                .cors().and()
                // Sessions disabled
                .sessionManagement().disable()
                // CSRF disabled (unnecessary with Bearer authentication)
                .csrf().disable()
                // Add CSP header
                .headers(headers -> headers
                        .contentSecurityPolicy(csp -> csp
                                .policyDirectives("default-src 'none' ; frame-ancestors 'none' ; sandbox")
                        )
                )// Security for HTTP requests enabled
                .authorizeHttpRequests(authorize -> authorize
                        // GET method for /api/v1/book is public
                        .requestMatchers(HttpMethod.GET, ...patterns: "/api/v1/book").permitAll()
                        // All other requests require authentication
                        .anyRequest().authenticated()
                )// Configuration for how to handle JWT authentication
                .oauth2ResourceServer(oauth2 -> oauth2.jwt()
                        .jwtAuthenticationConverter(jwtRoleAuthenticationConverter())
                ) // Add custom filters for audit logging and rate limiting
                .addFilterBefore(new RateLimitFilter(), ForceEagerSessionCreationFilter.class)
                .addFilterBefore(new AuditLoggingFilter(), BearerTokenAuthenticationFilter.class);
        return http.build();
    }
    // Converts JWT claims to Spring security GrantedAuthority
    public JwtAuthenticationConverter jwtRoleAuthenticationConverter() {
        JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();
        // Add authorities from roles claim
        grantedAuthoritiesConverter.setAuthoritiesClaimName("roles");
        // Add ROLE_ prefix for authorities
        grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");
        JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
        jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter);
        return jwtAuthenticationConverter;
    }
```

Liite 4. AuditLoggingFilter-luokka

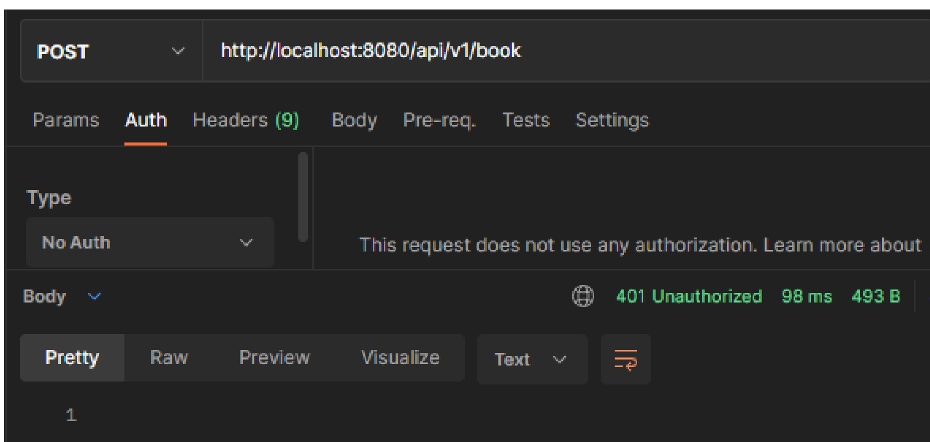
```
@Override
protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain
) throws ServletException, IOException {
    // Log request
    StringBuilder sbr = new StringBuilder("\nAuditLoggingFilter Pre:");
    sbr.append("\n     URI: ").append(request.getRequestURI())
            .append("\n     METHOD: ").append(request.getMethod())
            .append("\n     ORIGIN: ").append(request.getRemoteHost())
            .append("\n     TIME: ").append(LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
    // Write to file
    LogWriter.write(sbr.toString());
    // Call next filter of the chain
    filterChain.doFilter(request, response);
    // Log response
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    String challenge = response.getHeader( name: "www-authenticate");
    // Log entry base data
    StringBuilder sb = new StringBuilder("\nAuditLoggingFilter Post:");
    sb.append("\n     URI: ").append(request.getRequestURI())
            .append("\n     METHOD: ").append(request.getMethod())
            .append("\n     ORIGIN: ").append(request.getRemoteHost())
            .append("\n     TIME: ").append(LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME))
            .append("\n     STATUS: ").append(response.getStatus())
            .append("\n     USER: ").append(auth != null ? auth.getName() : null)
            .append("\n     AUTH: ").append(auth != null ? auth.getAuthorities() : null);
    // Log entry additional data on authentication failure
    if (challenge != null){
        sb.append("\n     MSG: ")
                .append(challenge.contains("Bearer error") ? challenge : "Bearer token not provided");
    }
    // Write to file
    LogWriter.write(sb.toString());
}
```

Liite 5. Autentikoinnin testit

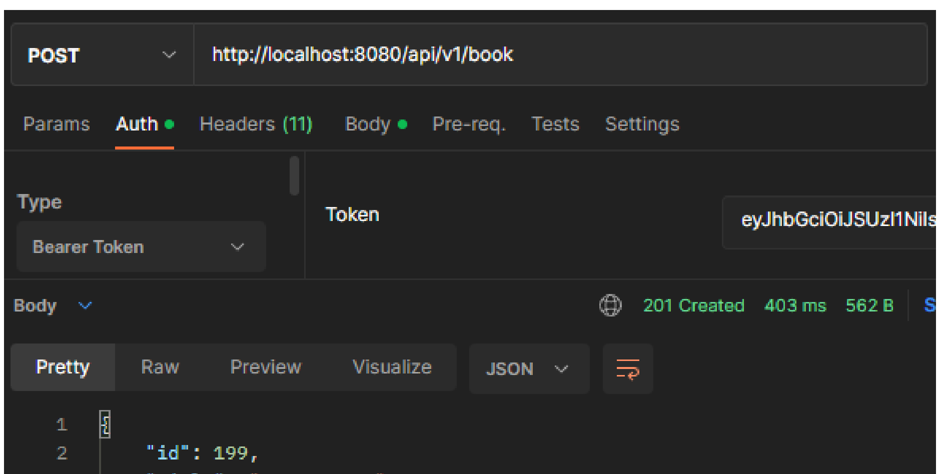
Testi 1 autentikoimaton GET-pyyntö osoitteeseen api/v1/book.



Testi 2 autentikoimaton POST-pyyntö osoitteeseen api/v1/book.



Testi 3 autentikoitu POST-pyyntö osoitteeseen api/v1/book.



Liite 6. Lokikirjauksen testit

Test 1 pyyntö osoitteeseen api/v1/book metodilla GET.

```
AuditLoggingFilter Pre:
     URI: /api/v1/book
     METHOD: GET
     ORIGIN: 0:0:0:0:0:0:0:1
     TIME: 2023-02-25T09:55:39.9779149

AuditLoggingFilter Post:
     URI: /api/v1/book
     METHOD: GET
     ORIGIN: 0:0:0:0:0:0:0:1
     TIME: 2023-02-25T09:55:40.15738
     STATUS: 200
     AUTH: [ROLE_ANONYMOUS]
     USER: null
```

Testi 2 pyyntö osoitteeseen api/v1/book metodilla POST, autentikoitu.

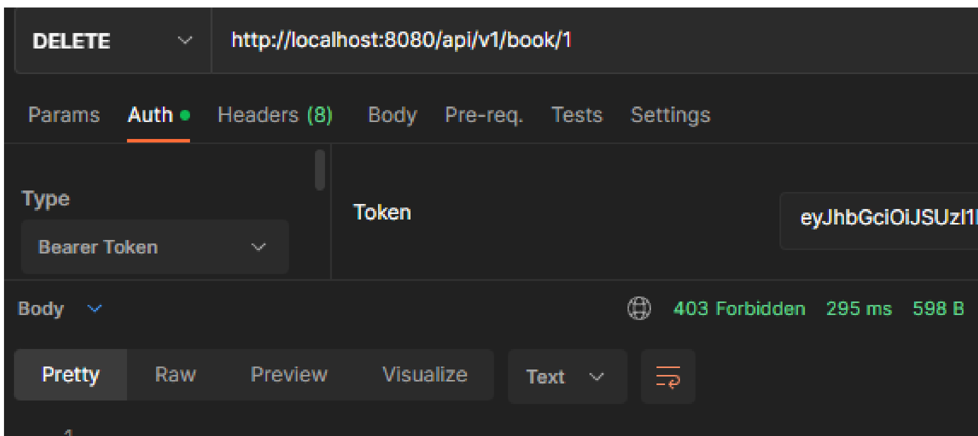
```
AuditLoggingFilter Pre:
     URI: /api/v1/book
     METHOD: POST
     ORIGIN: 0:0:0:0:0:0:0:1
     TIME: 2023-02-25T18:43:16.1026829

AuditLoggingFilter Post:
     URI: /api/v1/book
     METHOD: POST
     ORIGIN: 0:0:0:0:0:0:0:1
     TIME: 2023-02-25T18:43:16.4192946
     STATUS: 201
     USER: 76441ce5-0728-4cb5-90ef-52b100e882bc
     AUTH: [ROLE_user]
```

Testi 3 pyyntö osoitteeseen api/v1/book metodilla POST, virheellinen JWT.

```
AuditLoggingFilter Pre:
     URI: /api/v1/book
     METHOD: POST
     ORIGIN: 0:0:0:0:0:0:0:1
     TIME: 2023-02-25T18:43:30.90281

AuditLoggingFilter Post:
     URI: /api/v1/book
     METHOD: POST
     ORIGIN: 0:0:0:0:0:0:0:1
     TIME: 2023-02-25T18:43:30.9196225
     STATUS: 401
     USER: null
     AUTH: null
     MSG: Bearer error="invalid_token", error_description="An error occurred while attempting to decode the
Jwt: Malformed payload", error_uri="https://tools.ietf.org/html/rfc6750#section-3.1"
```

Liite 7. Auktorisoinnin testit

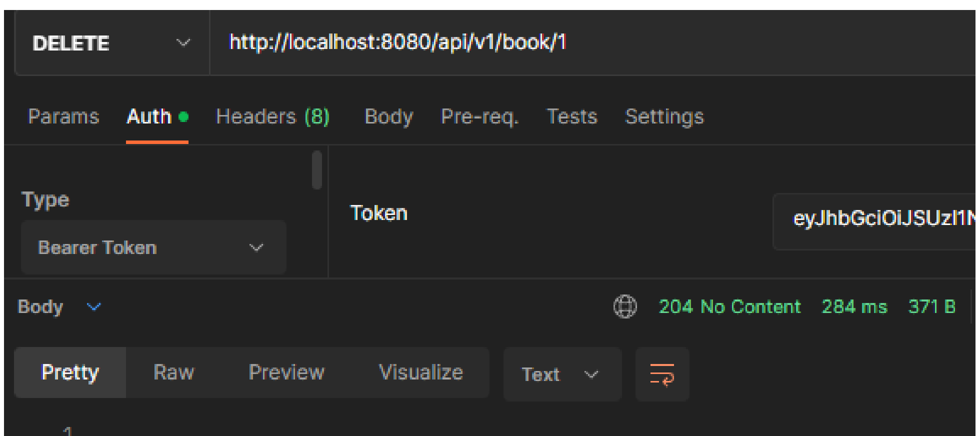
Test 1 pyyntö, jossa riittämätön käyttöoikeus.



Testi 1 lokikirjaus, kun pyynnöllä riittämätön käyttöoikeus.

```
STATUS: 403
USER: 76441ce5-0728-4cb5-90ef-52b100e882bc
AUTH: [ROLE_user]
MSG: Bearer error="insufficient_scope", error_
```

Testi 2 pyyntö, jossa riittävä käyttöoikeus.

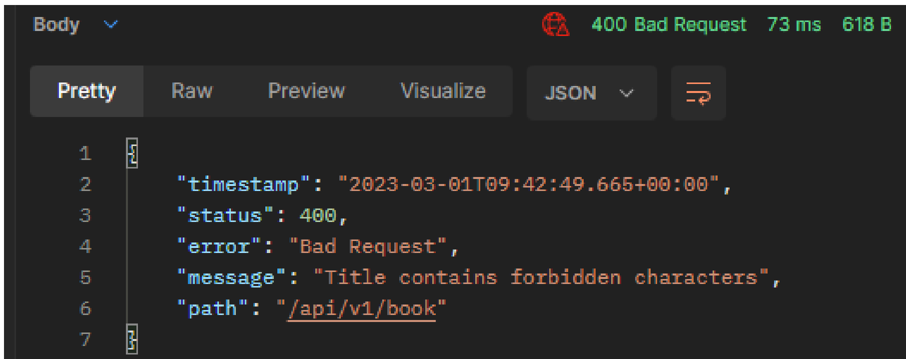


Testi 2 lokikirjaus, kun pyynnöllä riittävä käyttöoikeus.

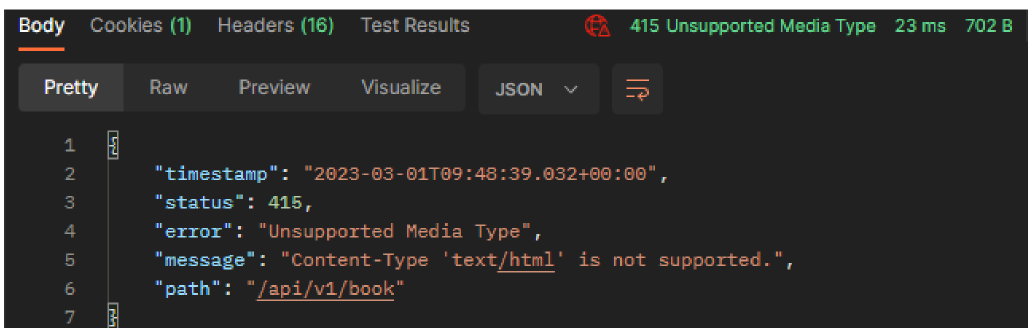
```
STATUS: 204
USER: 5c25a8fb-95ae-4d52-8bb4-f3f06b6c2427
AUTH: [ROLE_admin]
```

Liite 8. Validoinnin testit

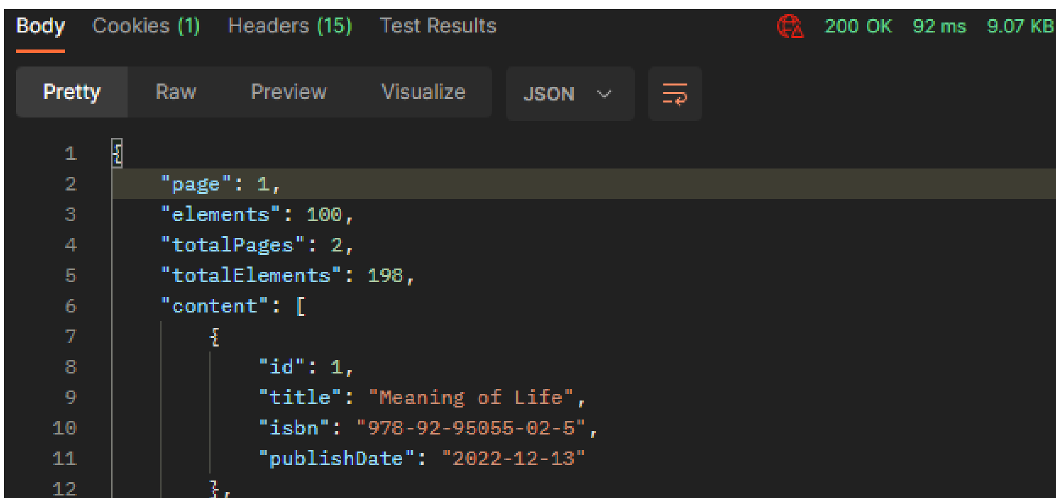
Testi 1 API:n vastaus, kun käyttäjän syöte ei vastaa sallittua.



Testi 2 API:n vastaus, kun käyttäjän syötteen mediatyyppi on väärä.



Testi 3 API:n vastauksen sisältämä datamäärä rajoitettu sivuttamalla.



Liite 9. Spring Security HTTP-otsikot

Spring Securityn oletusarvoiset tietoturvallisuutta lisäävät HTTP-otsikot.

X-Content-Type-Options	nosniff
X-XSS-Protection	0
Cache-Control	no-cache, no-store, max-age=0, must-revalidate
Pragma	no-cache
Expires	0
Strict-Transport-Security	max-age=31536000 ; includeSubDomains
X-Frame-Options	DENY