jamk.fi

Haavoittuvuusskannerit tietoturvallisessa sovelluskehityksessä

React-pohjaisen raportointikäyttöliittymän kehitys

Santeri Suihkonen

Opinnäytetyö
Huhtikuu 2020
Tekniikan ala
Insinööri (AMK), tieto- ja viestintätekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Suihkonen, Santeri	Opinnäytetyö, AMK	Huhtikuu 2020
	Sivumäärä 37	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Haavoittuvuusskannerit tietoturvallisessa sovelluskehityksessä

React-pohjaisen raportointikäyttöliittymän kehitys

Tutkinto-ohjelma

Tieto- ja viestintätekniikka, Insinööri (AMK)

Työn ohjaaja(t)

Esa Salmikangas, Jouni Huotari

Toimeksiantaja(t)

Organisaatio X

Tiivistelmä

Nykypäivänä tapahtuu kyberhyökkäyksiä enemmän kuin koskaan. Yksi yleisimmistä taustasyistä onnistuneelle hyökkäykselle on päivittämättömästä kohdejärjestelmästä löytyvät vakavat haavoittuvuudet. Nämä hyökkäykset keskimäärin koituvat hyvin kalliiksi ja niitä kannattaakin yrittää torjua aktiivisesti.

Tehtävänä ja tavoitteena oli kehittää toimeksiantajalle raportointikäyttöliittymä web-sovelluksena, jonka kautta voitaisiin nähdä ylläpidettävien verkkopalveluiden haavoittuvuustilanne. Kehitystyön ohella tutkittiin myös haavoittuvuusskannerien mahdollisia käyttötapauksia sovelluskehityksessä ja tuotantovaiheessa.

Kehittämistutkimuksena suoritettu työ tehtiin käyttäen laadullisen tutkimuksen menetelmiä. Tarpeellisen teorian keräämiseen ja sen avulla haavoittuvuusskannerien käytön ohjelmistokehityksen tukena määrittelyyn käytettiin laadullisen tutkimuksen menetelmiä.

Raportointikäyttöliittymä toteutettiin käyttäen React käyttöliittymäkirjastoa sekä Node.js palvelinta, joka hakee säännöllisesti skannausraporttien tuloksia haavoittuvuusskannereilta. Käyttöliittymä taas käyttää palvelinta REST-rajapinnan yli.

Lopputuotteena toimeksiantaja sai vaatimukset täyttävän web-sovelluksen, josta voidaan nähdä ylläpidossa oleviin palveluihin kohdistuvien haavoittuvuusskannausten kokonaiskuva, sekä tarkempia raportteja.

Avainsanat (asiasanat)

haavoittuvuusskanneri, haavoittuvuusskannaus, kyberturvallisuus, react, devsecops

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)
Suihkonen, Santeri

Type of publication
Bachelor's thesis

Date
April 2020

Language of publication:
Finnish

Number of pages
37

Permission for web publication: x

Title of publication
Vulnerability scanners in secure software development
Development of report UI using React

Degree programme
Information and Communications Technology

Supervisor(s)
Esa Salmikangas, Jouni Huotari

Assigned by
Organization X

Abstract

Today, more data breaches take place than ever before. One of the most common root causes for a successful data breach are critical vulnerabilities existing in unpatched computer systems. On an average, these data breaches end up being very expensive for the organization and should be prevented actively.

The assignment was to develop a reporting user interface in form of a web application for the Organization X. The UI should have the ability to show the current state of vulnerabilities in the software services administrated by the organization. In addition to the user interface, the use cases of vulnerability scanners during software development and in production phase were studied.

The assignment was carried out as a development study, and the software and all theory around it was studied using qualitative research methods. The use cases for usage of vulnerability scanners were also studied using qualitative methods.

The report user interface was developed using React library for the front-end and Node.js for the backend. The backend regularly fetches the results of vulnerability scans, which are then fetched to the front-end using the REST-API provided by the backend.

As a result, the Organization X got a web application, which met the set requirements. The application can be used to have an overall picture of the vulnerabilities in administrated software services. The user interface can also be used to read more specific vulnerability reports.

Keywords/tags (subjects)
vulnerability scanner, vulnerability scanning, cyber security, react, devsecops

Miscellaneous (Confidential information)

Sisältö

1 Johdanto ... 5

1.1 Lähtökohdat ... 5

1.2 Tutkimusasetelma ... 5

2 Haavoittuvuudet ... 7

2.1 Järjestelmähaavoittuvuudet ... 7

2.1.1 Haavoittuvuudet yleisesti ... 7

2.1.2 CVE ... 7

2.1.3 Haavoittuvuuksien vakavuuden mittarit ... 8

2.1.4 NVD ... 10

2.2 OWASP ... 10

2.3 Web-sovellushaavoittuvuudet ... 11

2.4 Yleisiä haavoittuvuusskannereita ... 11

2.4.1 ZAP ... 11

2.4.2 Nessus ... 12

2.4.3 Burp Suite ... 13

3 DevSecOps ... 13

3.1 DevOps ... 13

3.2 DevOps-prosessit ... 14

3.2.1 Versionhallinta ... 14

3.2.2 Jatkuva integraatio ... 14

3.2.3 Jatkuva julkaisu ... 15

3.3 DevSecOps ja haavoittuvuusskannerit ... 15

4 Käytetyt teknologiat ... 16

4.1 JavaScript ... 16

4.2 Node.js ... 17

4.3 React ... 18

4.3.1 Johdatus Reactiin ... 18

4.3.2 React-komponentit ... 18

4.3.3 Komponentin tila ja elinkaari ... 20

4.3.4 React Hooks ... 21
4.4 HTTP ... 22
4.4.1 HTTP yleisesti ... 22
4.4.2 Metodit ... 23
4.5 REST ... 24

5 Kehitystyö ... 25
5.1 Lähtötilanne ... 25
5.2 Suunnittelu ... 26
5.3 Toteutus ... 28
5.4 Tulokset ... 30

6 Pohdinta ... 33

Lähteet ... 35

Kuviot

Kuvio 1. Kehittämissyklin vaiheet ... 6
Kuvio 2. Yksinkertainen React-komponentti ... 18
Kuvio 3. React-sovelluksen juuri ... 19
Kuvio 4. Esimerkki Reactin propsista ... 19
Kuvio 5. Esimerkki tilallisesta laskurikomponentista ... 20
Kuvio 6. React Hookeilla luotu tila ... 22
Kuvio 7. Arkkitehtuurikuvaus ... 26
Kuvio 8. Käyttöliittymän päänäkymän mockup ... 27
Kuvio 9. Palvelukohtaisen näkymän mockup ... 28
Kuvio 10. Esimerkki palvelinpuolen HTTP-kyselystä ... 29
Kuvio 11. Palvelulistausnäkymä ... 30
Kuvio 12. Palvelukohtainen näkymä ... 31
Kuvio 13. Raporttinäkymä ... 32

Taulukot

Taulukko 1. CVSS 2.0 luokitukset 9
Taulukko 2. CVSS 3.x luokitukset 10

Lyhenneluettelo

AJAX	Asynchronous JavaScript and XML
DAST	Dynamic Application Security Testing
DOM	Document Object Model
GSA	U.S General Service Administration
JSX	JavaScript XML
LDAP	Lightweight Directory Access Protocol
RxJS	Reactive Extensions Library for JavaScript
SCAP	Security Content Automation Protocol
SQL	Structured Query Language

1 Johdanto

1.1 Lähtökohdat

Työn toimeksiantajana toimi kotimainen monialainen ohjelmistotalo, Organisaatio X. Toimeksiantona oli kehittää web-sovellus, joka toimii raportointikäyttöliittymänä haavoittuvuusskannerille. Käyttöliittymään tulisi voida kerätä haavoittuvuusskannauksien tai muiden testien tuloksia. Toimeksiannon täydennykseksi haluttiin tutkia haavoittuvuusskannerien käyttötapauksia sovelluskehityksen elinkaaren aikana.

Aihe oli myös ajankohtainen, sillä nykyään yksi yleisimmistä kyberrikollisten käyttämistä hyökkäysvektoreista on päivittämättömien järjestelmien sisältämät korjaamattomat tietoturva-aukot, eli haavoittuvuudet. Vakava haavoittuvuus on kuin avonainen ovi: hyökkääjän ei tarvitse hyökätä oven lukitusta, eli salasanaa vastaan, jos ovi on jo valmiiksi jätetty aukio. (Data Breach 101: Top 5 Reasons it Happens n.d.)

IBM:n vuonna 2016 suorittaman tutkimuksen mukaan yksittäinen tietoturvaloukkaus tai kyberhyökkäys maksaa hyökkäyksen uhriorganisaatiolle keskimäärin hieman alle neljä miljoona USA:n dollaria. Summa vaihtelee tietenkin hyökkäyksen laajuuden ja hyökkääjien saaman tiedon määrän ja laadun mukaan. (2016 Cost of Data Breach Study: Global Analysis 2016.)

Haavoittuvuuksien etsiminen sovelluksesta jo kehitysvaiheessa on tärkeää, koska niiden korjaaminen on silloin huomattavasti nopeampaa ja helpompaa, eli halvempaa, kuin jo julkaistussa sovelluksessa. Kehitysvaiheen korjaukset saapuvat myös varmasti loppukäyttäjille.

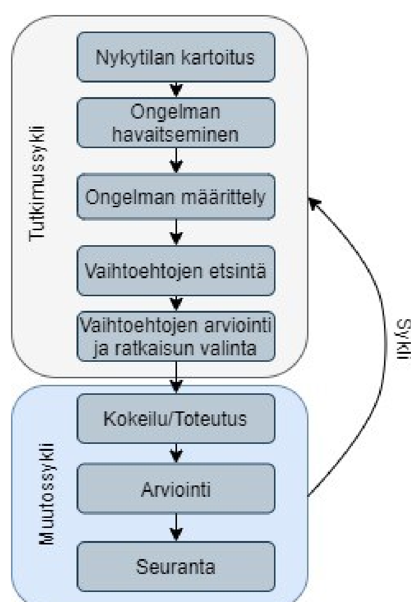
1.2 Tutkimusasetelma

Työn tavoiteltuna lopputuloksena oli kehittää toimeksiantajalle järjestelmä, jolla voidaan nähdä tuotannossa olevien palvelujen haavoittuvuustilanne. Sen lisäksi tutkittiin haavoittuvuusskannerien mahdollisia käyttökohteita sovelluskehitysprosessin eri

vaiheessa. Tutkimuskysymyksenä oli: ”Kuinka haavoittuvuusskannereita voidaan hyödyntää sovelluksen elinkaaren aikana sen tietoturvallisuuden parantamiseksi?”

Työ tehtiin kehittämistutkimuksena, koska työssä toteutettiin lopputuote toimeksiantajalle. Kehittämistutkimus voi olla yhdistelmä laadullista sekä määrällistä tutkimusta, mutta aivan yhtä hyvin se voi sisältää pelkästään laadullista tutkimusta. Laadullisessa tutkimuksessa on tarkoitus pyrkiä ymmärtämään tutkittavaa asiaa tai ilmiötä havainnoinnin ja haastattelujen kautta. Kehittämistutkimus on kuitenkin enemmän kuin laadullinen tai määrällinen tutkimus, koska siihen liittyy myös ongelman ratkaisu. (Kananen 2015, 34-42.)

Kehittämistutkimukselle ominaista on, että tutkimus suoritetaan kehittämissykleissä. Kuvio 1 on kuvattuna kehittämissyklin vaiheet. Yksittäisen syklin alussa aina kartoitetaan nykytila, eli etsitään ongelmakohta, jota halutaan kehittää. Tutkimussyklin aikana tutkitaan vaihtoehtoja, kuinka ongelma voitaisiin ratkaista. Kun vaihtoehdot ovat valmiina, arvioidaan ne tarpeen mukaan ja etsitään sopiva ratkaisu. Kun ratkaisu on löydetty, yritetään soveltaa sitä ongelmakohtaan. Tässä vaiheessa tutkimus siirtyy tutkimussyklistä muutossykliin. Kun tarvittavat muutokset ovat tehty, palataan taas tutkimussyklin arviointivaiheeseen. Kehittämistyö ja kehittämistutkimukset ovatkin usein jatkuvia prosesseja, jossa syklit toistuvat. (Kananen 2015, 41-42.)



Kuvio 1. Kehittämissyklin vaiheet (Kananen 2015, 42)

2 Haavoittuvuudet

2.1 Järjestelmähaavoittuvuudet

2.1.1 Haavoittuvuudet yleisesti

Haavoittuvuudella tarkoitetaan ohjelman logiikassa löytyvää virhettä tai heikkoutta, jota hyväksikäyttäen voidaan aiheuttaa vahinkoa järjestelmän tiedon luottamuksellisuudelle, eheydelle tai saatavuudelle. (Vulnerabilities n.d.a.) Haavoittuvuudet johtuvat usein suunnitteluvirheestä tai kehittäjän virheestä, kuten esimerkiksi riittämättömästä syötteiden tarkastuksesta (Vulnerabilities n.d.b).

Haavoittuvuuksia voidaan luokitella niiden hyväksikäyttötavan mukaan. Näistä tyypeistä yleisimpiä ovat haavoittuvuudet, jotka mahdollistavat hyökkääjän oman ohjelmakoodin suorittamisen, palvelunestohyökkäyksen tai puskuriylivuodon (Vulnerabilities By Type n.d.).

2.1.2 CVE

Haavoittuvuuksia tutkittaessa toistuu usein nimi CVE, joka on lyhenne sanoista Common Vulnerabilities and Exposures, joka tarkoittaa yleisiä haavoittuvuuksia ja konfiguraatio-ongelmia. CVE määrittelee sanan ”exposure” konfiguraatio-ongelmaksi tai virheeksi, joka ei suoraan anna hyökkääjälle pääsyä järjestelmään, kuten haavoittuvuus, mutta voi silti olla suuri hyökkäyksen vakavuutta edistävä tekijä. (Terminology 2017.)

CVE luotiin alun perin vuonna 1999, jolloin monet työkalut käyttivät haavoittuvuuksille pääasiassa omia tunnisteitaan. Tämä taas johti siihen, että työkalujen toimivuudessa oli suuria eroja eikä eri haavoittuvuustietokannat olleet toistensa kanssa yhteensopivia. MITRE ratkaisi tämän ongelman kehittämällä CVE standardin, jonka ansiosta työkalut ja haavoittuvuustietokannat voivat olla paremmin yhteensopivia toistensa kanssa. (About CVE 2019.)

CVE itsessään on lista, johon on eroteltu julkisesti tiedossa olevat järjestelmähaavoittuvuudet omien tunnistenimien alle siten, että jokaisella haavoittuvuudella on oma tunnisteensa. Tällainen tunniste on esimerkiksi CVE-2014-10001, joka sisältää haavoittuvuuden havaintovuoden, sekä uniikin numerosarjan. Tunnisteen lisäksi haavoittuvuudesta löytyy kuvaus sekä vähintään yksi julkinen maininta. (About CVE 2019.)

2.1.3 Haavoittuvuuksien vakavuuden mittarit

Haavoittuvuuksien vakavuuden luokitteluun käytetään CVSS eli Common Vulnerability Scoring System –mittaria. Nykyään jokaiselle uudelle julkisesti tiedossa olevalle CVE-haavoittuvuudelle annetaan pisteet 0-10 sen vakavuuden mukaan. Tällainen standardisoitu haavoittuvuuksien luokittelutapa soveltuu hyvin esimerkiksi organisaatioille sekä valtiollisille toimijoille sen yhtenäisyyden vuoksi. Tällaista mittaria voidaan käyttää esimerkiksi korjausten kehityksen priorisoinnin tukena. Tarkasteluhetkellä CVSS-järjestelmästä on yleisessä käytössä eri versioita: versiot 2.0 ja 3.x, ja sen uusin versio on 3.1. (Vulnerability Metrics n.d.)

Haavoittuvuuksien vakavuutta mitattaessa otetaan huomioon hyökkäysvektori, eli voidaanko hyökkäys suorittaa verkon yli, tarvitaanko pääsy esimerkiksi samaan lähiverkkoon kohteen kanssa, vai täytyykö kohteeseen olla fyysinen pääsy. Myös suoritettavan hyökkäyksen monimutkaisuus vaikuttaa pisteytykseen. Hyökkäys, joka voidaan toistaa ilman mitään reunaehtoja, on vakavampi kuin hyökkäys, jonka suoritusta varten täytyy tehdä paljon ennakkotyötä. Myös se, tarvitseeko esimerkiksi kohdejärjestelmän käyttäjän tehdä jotakin hyökkäyksen onnistumiseksi vaikuttaa vakavuuteen. (Common Vulnerability Scoring System v3.1: Specification Document n.d.)

Yleisesti vakavuutta laskettaessa otetaan huomioon hyökkäyksen vaikutusten laajuus. Esimerkiksi jos hyökkäyksen kohteena on haavoittuvainen virtuaalikone ja hyökkääjä pystyy siinä esiintyvää haavoittuvuutta hyödyntäen muokkaamaan isäntäkoneen tiedostoja. Tällöin haavoittuvuus on vakavampi kuin sellainen, jonka vaikutukset ulottuvat vain kyseiseen virtuaalikoneeseen. (Common Vulnerability Scoring System v3.1: Specification Document n.d.)

Tärkeitä mittareita ovat myös vaikutukset kohteen luottamuksellisuuteen, eheyteen ja saatavuuteen. Näihin kohdistuvat vaikutukset voidaan luokitella olemattomaksi, osittaiseksi tai kokonaiseksi luottamuksellisuuden, eheyden tai saatavuuden menetykseksi. Jos hyökkäyksen vaikutukset ulottuvat laajemmalle, kuin itse haavoittuvaiseen järjestelmään, lasketaan pisteet pahimman alueen mukaan. (Common Vulnerability Scoring System v3.1: Specification Document n.d.)

Muita tekijöitä haavoittuvuuden vakavuuden määritykseen ovat hyökkäyskoodin saatavuus, haavoittuvuuden korjauksien taso, sekä kuinka tarkasti haavoittuvuus on raportoitu. Esimerkiksi usea julkisesti saatavilla oleva käyttövalmis hyökkäysskripti ja kattavasti teknisellä tasolla raportoitu haavoittuvuus nostaa sen tasoa. Jos haavoittuvuudelle ei ole olemassa toimivaa korjauspäivitystä, sen taso on myös korkeampi. (Common Vulnerability Scoring System v3.1: Specification Document n.d.)

Nykyisin käytössä olevasta kahdesta versiosta versio 2.0 sisältää kolme vakavuusluokkaa: Matala, keskitaso ja korkea. Versio 3.x puolestaan sisältää viisi vakavuusluokkaa: Olematon, matala, keskitaso, korkea ja kriittinen. Vakavuusluokat määritellään haavoittuvuuden saamien vakavuuspisteiden mukaan taulukoiden 1 ja 2 mukaisesti. (Vulnerability Metrics n.d.)

Taulukko 1. CVSS 2.0 luokitukset

Vakavuus	Pisterajat
Matala	0.0-3.9
Keskitaso	4.0-6.9
Korkea	7.0-10

Taulukko 2. CVSS 3.x luokitukset

Vakavuus	Pisterajat
Olematon	0.0
Matala	0.1-3.9
Keskitaso	4.0-6.9
Korkea	7.0-8.9
Kriittinen	9.0-10.0

2.1.4 NVD

NVD eli National Vulnerability Database on NISTin ylläpitämä tietokanta. Sinne on kerätty pääasiassa CVE-haavoittuvuuksia, mutta sieltä löytyy myös esimerkiksi tietoturvallisuuden tarkastuslistoja, konfiguraatio-ongelmia sekä tuotteiden nimiä. Tietokanta on luotu esittämään tietoa SCAP:n mukaisesti, joka mahdollistaa haavoittuvuustietojen käytön ristiin eri työkaluissa ja esimerkiksi haavoittuvuuksien hallinnan automatisoinnin. (General Information n.d.)

2.2 OWASP

Vuonna 2001 perustettu OWASP eli The Open Web Application Security Project on voittoa tavoittelematon järjestö, jonka tavoitteena on luoda ilmaista materiaalia sovellustietoturvan parantamiseksi. OWASP:n luomiin materiaaleihin kuuluu muun muassa dokumentaatiota, työkaluja, opetusympäristöjä, ohjesäännöksiä sekä tarkastuslistoja. (About The Open Web Application Security Project 2019.)

OWASP:n tärkeimpiin, niin sanottuihin lippulaivaprojekteihin kuuluu esimerkiksi ZAP tietoturvatestaustyökalu, jota tutkitaan tarkemmin luvussa 2.4.1. Muita tärkeitä OWASP:n projekteja ovat esimerkiksi Cheat Sheets, Top Ten ja Application Security Verification Standard. Cheat Sheets sisältää sovelluskehittäjille ohjeistuksia tietoturvalliseen kehitykseen, Top Ten listaa yleisimpiä haavoittuvuuksia web-sovelluksista ja

Application Security Verification Standard tarjoaa kattavamman standardin sovelluskehitykseen.

2.3 Web-sovellushaavoittuvuudet

Web-sovelluksiin, eli selaimella käytettäviin ohjelmistoihin liittyy usein tietyn tyyppisiä haavoittuvuuksia. Näistä haavoittuvuuksista yleisimmät ja oleellisimmat on kerätty dokumentoituna OWASP Top Ten –listaan. Viimeisimmän Top Ten-listan mukaan yleisin tietoturvaongelma on edelleen injektiohaavoittuvuus, joka pitää sisällään SQL-, NoSQL-, käyttöjärjestelmä- sekä LDAP-injektiot. Muita yleisiä ongelmia ovat esimerkiksi puutteellinen autentikointi tai pääsynhallinta ja niin sanottu cross-site scripting, jossa hyökkääjä onnistuu injektoimaan omaa koodiansa web-sovellukseen. (OWASP Top 10 – 2017: The Ten Most Critical Web Application Security Risks 2017.)

Tarkempana ohjeistuksena sovelluskehityksen yhteydessä voidaan käyttää OWASP Application Security Verification Standardia, joka tunnetaan myös lyhenteellä ASVS. ASVS toimii kattavana linjauksena web-sovellusten kehityksen jokaisen vaiheen tietoturvallisuuden parantamiseksi. OWASP suosittelee ASVS:n käyttöä mittarina sovelluksen tietoturvan tasolle, ohjelinjauksena tai apuna tietoturvavaatimusten luontiprosessissa. Siihen on kasattu tarkkoja testitapauksia, jotka on suunnattu toimintaohjeeksi niin ohjelmistokehittäjille kuin tietoturvatestaajillekin. (OWASP Application Security Verification Standard n.d.)

2.4 Yleisiä haavoittuvuusskannereita

2.4.1 ZAP

OWASP ZAP eli OWASP Zed Attack Proxy on OWASP:n avoimen lähdekoodiin perustuva vapaaehtoisten kehittämä yksi maailman tunnetuimmista haavoittuvuusskannereista. Sillä voi etsiä automaattisesti haavoittuvuuksia web-sovelluksista, eikä se ole kaupallinen, joten se on täysin jokaisen vapaassa käytössä. (OWASP ZAP n.d.)

ZAP:lla voidaan kartoittaa web-sovellusta painamalla automaattisesti jokaista linkkiä, joka web-sivulta löytyy, sekä seuraamalla sovelluksen tekemiä AJAX-kutsuja. ZAP toimii välityspalvelimena, jolloin kaikki haluttu verkkoliikenne kulkee sen kautta. ZAP passiivisesti tutkii käyttäjän verkkosivulle lähettämiä kyselyitä mahdollisten haavoittuvuuksien osalta. Lisäksi siitä löytyy myös perinteinen ”kovaääninen” haavoittuvuusskannausominaisuus, jolla voidaan tehokkaasti löytää haavoittuuvuuksia. (ZAP API Documentation n.d.)

ZAP on automatisoinnin kannalta hyvä vaihtoehto, koska siinä on myös kattava REST API, jolla sen toimintoja voidaan kutsua verkon yli (ZAP API Documentation n.d.). Tämä mahdollistaa skannauksien automatisoinnin esimerkiksi sovelluskehitysprosesseissa tai ajastetut skannaukset tuotantovaiheessa.

2.4.2 Nessus

Maailman käytetyin haavoittuvuusskanneri, eli Nessus on kaupallinen Tenablen kehittämä ratkaisu. Nessuksella voidaan määrittää tapauskohtaisesti, millä tasolla kohdetta halutaan tarkastella. Sillä voidaan suorittaan yksinkertainen porttiskannaus tai päinvastoin ajaa kohteeseen kaikki mahdolliset testit. Näitä yksittäisiä testitapauksia kutsutaan Nessuksessa plugineiksi, joita on kirjoitushetkellä yli 137000. Pluginit voivat testata kohteesta esimerkiksi CVE-löydöksiä, vanhentuneita sovellusversioita tai konfiguraatiovirheitä. (The Nessus Family 2019.)

Nessuksesta on tarjolla kolme erilaista versiota: Nessus Essentials, Nessus Professional sekä Tenable.io. Essentials on ilmainen, mutta myös rajattu esimerkiksi samanaikaisten skannauksien osalta määrä on rajattu 16 IP-osoitteeseen kerralla. Essentials on suunnattu lähinnä opiskelijoille- ja harrastuskäyttöön. Nessus Professional -versiossa ei ole rajoituksia samanaikaisille skannauksille. Suuremmille organisaatioille suunnattu Tenable.io on pilvipohjainen ympäristö, josta voidaan hallita Nessus-skannereita. (The Nessus Family 2019.) Tenable.io sisältää myös kattavan REST-sovellusrajapinnan, mitä voidaan käyttää haavoittuvuustestauksien automatisoinnissa. Rajapintaa voidaan käyttää halutessaan täysin manuaalisesti, mutta sitä varten

on myös Java ja Python ohjelmointikielille tehdyt viralliset kirjastot. (Get Started 2020.)

2.4.3 Burp Suite

Kaupallinen PortSwiggerin kehittämä Burp Suite on monikäyttöinen työkalu web-sovellusten tietoturvatestaukseen. Sitä voidaan käyttää manuaalitestauksessa välityspalvelimena, jonka avulla voidaan pysäyttää HTTP-kyselyitä ja muokata niitä ennen, kuin ne lähtevät eteenpäin kohteeseen testaajan tietokoneelta. Kyselyitä voidaan lähettää uudestaan muokattuina tai suorittaa hyökkäyksiä lähettämällä iso määrä kyselyjä määritellyillä arvoilla. Burp Suitessa on mukana myös haavoittuvuusskanneri, joka tunnistaa yleisimpiä turvallisuusongelmia web-sovelluksista. (The Burp Suite Family 2020.)

Burp Suitesta on kolme erilaista versiota: Community, Professional ja Enterprise. Nämä versiot ovat keskenään melko erilaisia. Community-versio on suunnattu harrastelijoille ja on ilmainen, mutta siinä on vain välttämättömimmät manuaaliset työkalut. Professional-versio on suunnattu ammattilaiskäyttöön esimerkiksi penetraatiotestaajille. Siinä on mukana manuaaliset työkalut, sekä automaattisia työkaluja, joilla voidaan suorittaa automatisoituja hyökkäyksiä web-sovelluksiin. Enterprise -versio on täysin automatisoitavissa oleva haavoittuvuusskanneri, joka voidaan integroida sovelluskehitysprosesseihin. (The Burp Suite Family 2020.)

3 DevSecOps

3.1 DevOps

DevOps nimi tulee sanoista development ja operations. Tämä tarkoittaa, että kehittäjät (eng. developers) ja palvelinylläpito (eng. operations) tekee tiivistä yhteistyötä. Devops tuokin tuotannossa olevat palvelut lähemmäksi näiden kehittäjiä. Devopsin tavoitteena on automatisoida lähes kaikki sovelluksen tuotantoon viemiseen vaadit-

tavat prosessit siitä, kun kehittäjä työntää valmiin ominaisuuden versiohallintaan, siihen kun tämä ominaisuus on tuotantoversiossa mukana. Näitä prosesseja voivat olla koostaminen (eng. build), laadunvarmistus ja julkaisuprosessit. (Klemetti 2013.)

3.2 DevOps-prosessit

3.2.1 Versionhallinta

Versionhallinnasta puhuttaessa tarkoitetaan ohjelmistoa, joka tallentaa tiedostojen muokkaushistorian, jolloin aiempiin versioihin voidaan palata myöhemmin. Versionhallintaa käytetään usein nimenomaan ohjelmakoodin yhteydessä, mutta sitä voidaan käyttää kaikenlaisten tiedostojen muutosten seurantaan. (Chacon & Straub 2014, 1.1.) Yleisin esimerkki versionhallinnasta on Git-versionhallinta.

DevOps-kehitystavassa automatisoidut operaatiot alkavat siitä, kun kehittäjä työntää muutoksensa versionhallintaan. Kun versionhallintaan uuden ominaisuuden puskemisen yhteydessä sovellus koostetaan ja sitä vasten suoritetaan testejä, koko prosessia kutsutaan jatkuvaksi integraatioksi. (What is Continuous Integration? n.d.)

3.2.2 Jatkuva integraatio

Jatkuvan integraation perusperiaatteena on, että jokainen kehittäjän tuottama uusi koodi testataan ongelmien varalta. Yleisesti käytetty avoimen lähdekoodin tuote sovelluksen koostamiseen on esimerkiksi Jenkins. Sovellus koostetaan ja siihen suoritetaan esimerkiksi yksikkötestit, joista näkee heti, toimiiko kehittäjän tuottama koodi oikein.

Tässä vaiheessa voidaan suorittaa muitakin testejä, kuten koodin analysointia. Koodin laatua voidaan analysoida automaattisesti staattisella skannerilla, kuten esimerkiksi SonarQubella (Source Code Analysis Tools n.d.). Staattiset analysointityökalut ovat siltä osin hyviä, että niitä voidaan käyttää pelkkään lähdekoodiin ilman toimivaa sovellusta.

3.2.3 Jatkuva julkaisu

Jatkuva julkaisu on käännös englannin termeistä continuous deployment tai continuous delivery. Jatkuva julkaisu tarkoittaa, että uusin versio palvelusta tuodaan käyttäjille käyttöön jatkuvasti ja säännöllisesti, esimerkiksi vaikka päivittäin. Jokainen julkaisu tehdään automaattisen jatkuvan integraatioputken läpi. Jatkuva julkaisu tarvitseekin kattavat testiautomaatiot, jotta voidaan varmistaa jokaisessa julkaisussa olevan mahdollisimman vähän virheitä. (Huotarinen 2016.)

3.3 DevSecOps ja haavoittuvuusskannerit

Sovelluskehityksen operaatioiden automatisoinnin lisäksi on tärkeää, että tietoturvallisuus otetaan huomioon jokaisessa sovelluskehityksen vaiheessa suunnittelusta tuotantoon. DevSecOpsista voidaan puhua, kun tietoturvallisuus ei ole erillinen kokonaisuus, vaan kulkee mukana DevOps-kehityksen alusta asti. Tällä voidaan tarkoittaa esimerkiksi DevOps putkeen asetettavia automaattisia tietoturvatestauksia, kehittäjien tietoturvallisuuskoulutuksia sekä kehitystiimien välistä läpinäkyvyyttä. Olennaista DevSecOps -ajattelussa on kuitenkin se, että koko kehitystiimi ottaa tietoturvallisuuden huomioon. (What is DevSecOps n.d.)

Iso osa DevSecOpsia on haavoittuvuusskannausten automatisointia sekä tietoturvallisuuden rajojen miettimistä kehitettävän sovelluksen ja sen sisältämän mahdollisen tiedon luonteen mukaan – kuinka kovia tietoturvakontrolleja sovellus oikeasti tarvitsee (What is DevSecOps n.d.)? Kehitteillä olevaa sovellusta voidaan testata automaattisesti esimerkiksi aiemmin mainituilla skannereilla eli Nessuksella, ZAPilla tai Burp Suiten Enterprise-versiolla. Näitä skannauksia voitaisiin ajaa DevOps-prosessien mukaisesti joko suoraan kehittäjän versionhallintaan tekemän muutoksen seurauksena tai säännöllisin väliajoin tehtynä. Itse kehitysprosessin tukena kehittäjät voivat käyttää esimerkiksi Burp Suitea kehitettävien ominaisuuksien tietoturvan testaukseen ja jatkuvaan parantamiseen.

Kehitys ei suinkaan lopu siinä vaiheessa, kun sovellus julkaistaan tuotantoon. Tuotannossa olevista sovelluksista voi aina löytyä uusia ongelmia monitoroinnin yhteydessä.

GSA on listannut esimerkiksi Nessuksen sopivaksi haavoittuvuuksien monitorointityökaluksi. (Building a DevSecOps Culture - from a Technical Perspective n.d) Jos tuotannossa olevasta sovelluksesta löydetään uusi haavoittuvuus, jonka olemassaolo mahdollistaa hyökkäyksen sovelluksen tai sen ympäristön sisältämän tiedon luottamuksellisuuteen, eheyteen tai saatavuuteen, tulee palata suunnittelupöydälle ja miettiä kuinka kyseisen haavoittuvuuden kanssa toimitaan.

Tästä päästään takaisin tutkimuskysymykseen, eli kuinka haavoittuvuusskanneria voisi hyödyntää sovelluksen elinkaaren aikana? Kehitysvaiheessa voidaan automaattisten skannauksien, sekä kehittäjien omien testausten perusteella parantaa sovelluksen tietoturvan laatua nopeasti. Pelkkä skannausraportti ei kuitenkaan itsessään sovelluksen tietoturvaa paranna, vaan kehitystiimin täytyy osata reagoida tuloksiin tilanteeseen sopivalla tavalla.

Tuotantovaiheessa säännöllisillä skannauksilla voidaan löytää uusia haavoittuvuuksia sovelluksen taustateknologioista tai itse sovelluksesta, ja korjata ne ennen kuin hyökkääjät ehtivät käyttää niitä hyväksi. Näiden säännöllisten skannauksien hallinta voidaan etenkin suuremmissa ympäristöissä joskus kokea hankalaksi, jolloin voidaan kehittää omia raportointiratkaisuja hyödyntäen haavoittuvuusskannereiden sovellusrajapintoja.

4 Käytetyt teknologiat

4.1 JavaScript

JavaScript on kevyt, suoritushetkellä tulkittava ohjelmointikieli, joka on alun perin luotu web-sivujen toiminnallisuuden toteuttamiseksi. Nykyään JavaScriptiä voidaan kuitenkin suorittaa myös selaimen ulkopuolella eri ohjelmistojen avulla. Näitä ohjelmistoja ovat esimerkiksi Node.js, Apache CouchDB ja Adobe Acrobat. JavaScriptiä ei pidä myöskään sekoittaa tunnettuun Java-ohjelmointikieleen, vaikka nimissä onkin yhtäläisyyksiä. (JavaScript 2020.)

JavaScriptin ominaisuudet perustuvat ECMAScript standardiin ja kaikkien nykyaikaisten selaimia pitäisi noudattaa ainakin ECMAScript 5.1 standardia. Vuodesta 2015 eteenpäin ECMAScript standardien versiot numeroidaan julkaisuvuoden mukaan ja uusi versio julkaistaan vuosittain. Uusin versio tarkasteluhetkellä on ECMAScript 2020. (JavaScript 2020.)

4.2 Node.js

JavaScript on perinteisesti ollut vain selaimessa suoritettava ohjelmointikieli verkkosivujen toiminnallisuuden luontia varten. Vuonna 2009 tähän kuitenkin tuli muutos nimeltä Node.js, eli JavaScriptin suoritusympäristö palvelimella. Koska Node.js mahdollistaa, että molemmat, käyttöliittymä sekä palvelinpuoli käyttää JavaScriptiä, monet käyttöliittymäkehittäjät pystyvät matalalla kynnyksellä osallistumaan myös palvelinpuolen kehitykseen opettelematta uusia työkaluja. (Copes 2018, 6-9.)

Node.js on nykyään avoimeen lähdekoodiin perustuva JavaScriptin suoritusympäristö, joka käyttää samaa Googlen V8 JavaScript-moottoria kuin esimerkiksi Chrome -selain. Node.js applikaatio on aina yksi prosessi, jossa ei sen vuoksi mene resursseja hukkaan säikeiden luontiin. Node.js ei siis käytä säikeistystä lainkaan. Node.js kirjastofunktiot on kirjoitettu pääasiassa asynkronisesti, joten yhdenkään palvelinkutsun ei pitäisi sulkea sovelluksen muuta toimintaa suorituksensa ajaksi. Säikeistyksen puuttuminen helpottaa myös kehittäjien työtä, sekä mahdollisesti vähentää siitä johtuvien ohjelmistovirheiden määrää. (Copes 2018, 6-9.)

Itsessään Node.js on vain ympäristö, jossa suoritetaan JavaScriptia, mutta Node.js:lle on myös erittäin kattava määrä kirjastoja, joita voi ottaa käyttöön Noden pakettienhallintaohjelmalla, eli npm:llä. Yksi esimerkki näistä kirjastoista on web-palvelinkehys Express, jota käytettiin toimeksiantajalle kehitetyn sovelluksen palvelimen sovelluskehyksenä. (Copes 2018, 8-10.)

4.3 React

4.3.1 Johdatus Reactiin

React on JavaScript-pohjainen sovelluskehys (eng. framework), joka parhaiten soveltuu interaktiivisten käyttöliittymien toteutukseen. Reactia kehittää Facebook yhdessä kehittäjäyhteisön kanssa. Reactin käyttö perustuu uudelleenkäytettävien komponenttien luomiseen. (React 2019.)

React käyttää komponenteissaan syntaksia, joka näyttää HTML-kieleltä, mutta on oikeasti JavaScriptin syntaksin laajennus. Tämän syntaksin nimi on JSX ja se on olemassa vain helpottaakseen kehittäjien työtä tekemällä koodista luettavampaa. JSX:n avulla voidaan upottaa ”HTML”-elementteihin aaltosulkujen sisälle JavaScriptiä, jota nähdään luvun 4.3.2 esimerkeissä. (Introducing JSX 2020.)

4.3.2 React-komponentit

Reactilla kehitetty käyttöliittymä koostuu Reactin omista ”rakennuspalikoista” eli komponenteista. Yksinkertaisimmillaan tällainen komponentti on JavaScript funktio, joka palauttaa JSX-elementin, kuten Kuvio 2 esimerkissä palautetaan div-elementti.

```
const ExampleComponent = () => {
  return (
    <div>Hello!</div>
  )
}
```

Kuvio 2. Yksinkertainen React-komponentti

Jokaiselle React-sovellukselle täytyy määrittää ainakin yksi *ReactDOM.render()*-funktio, jolla React-komponentti piirretään DOMiin. Tähän funktioon määritetään ensimmäiseksi parametriksi React-komponentti, joka halutaan piirtää funktion

toisena parametrina annettavaan HTML-elementtiin. Kuvio 3 esimerkkikoodissa ExampleComponent piirrettäisiin sivun HTML-elementtiin, jonka id on ”root”.

```
ReactDOM.render(
  <ExampleComponent />,
  document.getElementById('root')
);
```

Kuvio 3. React-sovelluksen juuri

React on suunniteltu siten, että sillä voidaan luoda uudelleenkäytettäviä komponentteja. Kun käyttöliittymään halutaan luoda esimerkiksi jonkinlaista listaa, voi tulla vastaan tilanne, jossa tarvitaan keskenään rakenteeltaan samanlaisia komponentteja, joiden sisältämä tieto on kuitenkin toistensa välillä erilaista. Tätä lopputulosta hakiessa voidaan käyttää komponentin ”propseja”. Kuvion 4 esimerkissä käytetään propseja message-arvon määrittämiseen. Arvo annetaan ExampleComponentin palautusarvon ExampleChildComponentin kutsussa.

```
const ExampleComponent = () => {
  return (
    <ExampleChildComponent message={'world'} />
  )
}

const ExampleChildComponent = (props) => {
  return (
    <div>Hello {props.message}</div>
  )
}
```

Kuvio 4. Esimerkki Reactin propsista

4.3.3 Komponentin tila ja elinkaari

Komponentti voi olla funktiopohjainen, kuten aiemmat esimerkit ovat olleet. Tällainen funktiopohjainen komponentti ei kuitenkaan voi perinteisesti käyttää Reactin elinkaarimetodeja eikä tilaa. Näitä ominaisuuksia käytettäessä täytyy tehdä komponentista JavaScript-luokka ja periä se React.Component-luokasta. (React.Component 2020.) Tämä kuitenkin muuttui vuoden 2019 helmikuussa Reactin 16.8 versiopäivityksessä, jossa tuotiin uutena ominaisuutena React Hooks, joihin syvennytään tarkemmin kappaleessa 4.3.4.

Kun komponentissa halutaan esittää sellaista tietoa, jota halutaan päivittää, voidaan käyttää apuna komponentin tilaa. Kuvio 5 esittää tilallista laskurikomponenttia, josta nähdään, kuinka tila alustetaan luokan *constructor*-metodissa, joka on myös React-komponentin elinkaaren ensimmäinen metodi. Tämän jälkeen React kutsuu luokan *render*-metodia. Renderin jälkeen kutsutaan *componentDidMount*-metodia, jossa suoritetaan sellaista koodia, joka täytyy suorittaa siinä vaiheessa, kun tiedetään että komponentti on varmasti valmis. Esimerkiksi RxJS-kirjaston *subscribe*-metodi on hyvä esimerkki tästä tapauksesta. Esimerkissä nähdään myös, kuinka yksinkertaisen JSX button-elementin *onClick*-tapahtumassa kutsutaan *increment*-metodia, jossa tilaa muokataan *setState*-metodilla. (React.Component 2020.)

```
class ExampleClassComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      counter: 0
    }
  }
  componentDidMount() {}

  increment = () => {
    this.setState({
      counter: counter + 1
    })
  }
  render() {
    <div>
      <button onClick={() => this.increment()}>Add one</button>
      Times clicked: {this.state.counter}</div>
  }
}
```

Kuvio 5. Esimerkki tilallisesta laskurikomponentista

Kun tilaa muokataan tai komponentin props-objekti muuttuu, komponentti päivitetään. Päivitettäessä React kutsuu ensin komponentin *render*-metodia, jonka jälkeen kutsutaan *componentDidUpdate*-metodia. Tässä metodissa voidaan kutsua sellaista koodia, jota halutaan suorittaa aina komponentin päivittyessä, kuten esimerkiksi AJAX-kyselyjä. Kun taas komponenttia ei enää tarvita ja se hävitetään käyttöliittymästä, kutsutaan *componentWillUnmount*-metodi. Tässä metodissa on tarkoitus siivota esimerkiksi kaikki käynnissä olevat ajastimet tai verkkokyselyt. (React.Component 2020)

4.3.4 React Hooks

Reactin versiosta 16.8 eteenpäin on voinut käyttää ominaisuutta nimeltä Hooks. Tämä on tapa käyttää komponentin tilaa ja elinkaareen liittyviä metodeja funktiopohjaisessa komponentissa. Syy Hookien kehityksen taustalla johtui luokkapohjaisten komponenttien monimutkaisuudesta ja miten ei-toisiinsa liittyviä asioita täytyi suorittaa samojen elinkaarimetodien aikana. Luokkapohjaisia komponentteja ei kuitenkaan ole tarkoitus poistaa Reactista tulevaisuudessa. Hookkeja on vakiona kahdenlaisia: efekti- ja tilahookkeja.

Tilahookilla voidaan käyttää tilaa funktiokomponentissa, josta esimerkkinä on Kuvio 6 esiintyvä sama laskurikomponentti kuin kappaleen 4.3.3 esimerkkikuvassa. Hookeilla toteutettu komponentti on vain selkeämpi lukea, koska siinä ei tavitse esimerkiksi käyttää *this*-määritettä. Tilan käyttö *useState*-hookilla toimii määrittelemällä tilan ominaisuuden nimi ja sen päivitysmetodin nimi, esimerkissä *counter* ja *setCounter*. Tilalle asetetaan myös aloitusarvo useState-funktion parametriksi.

```
const ExampleHookComponent = (props) => {
  const [counter, setCounter] = useState(0)

  const increment = () => {
    setCounter(counter + 1)
  }
  return (
    <div>
      <button onClick={() => increment()}>Add one</button>
      Times clicked: {counter}
    </div>
  )
}
```

Kuvio 6. React Hookeilla luotu tila

4.4 HTTP

4.4.1 HTTP yleisesti

Lähes jokaisen internetselaimella tehdyn liikkeen takana on HTTP, eli Hypertext Transfer Protocol. Alkujaan ensimmäinen virallinen versio HTTP/0.9 luotiin vuonna 1991 ja pitkään laajassa käytössä ollut versio HTTP/1.1 (RFC 2616) spesifikaation viimeisin versio julkaistiin vuonna 1999. HTTP/1.1 on edelleen käytössä monissa verkkopalveluissa. (Zalewski 2011, 41-42.)

Nykyään uusin laajassa käytössä oleva HTTP-versio on HTTP/2.0 (spesifikaatio RFC 7540). Versio 2.0 ei muuta protokollan peruselementtejä mitenkään, joten sitä pystyy käyttämään samalla tavalla, kuin versiota 1.1. HTTP/2.0 on kuitenkin tehty suorituskykyisemmäksi, kuin vanhemmat versiot. (RFC 7540:2015, 1)

HTTP perustuu TCP-yhteyteen. TCP-protokollalla voidaan varmistaa, että tieto saapuu kokonaisena määränpäähänsä. Kun TCP-yhteys on muodostettu, voidaan web-

palvelimelta pyytää tietoja. HTTP/1.1 ja 2.0 kyselyyn määritellään metodi, eli kerrotaan mitä halutaan tehdä, suhteellinen polku haluttuun resurssiin, HTTP-protokollan versio, sekä Host-otsake, jolla kerrotaan miltä palvelimelta (URL-osoitteesta) tieto haetaan. Näiden lisäksi voidaan määritellä valinnaisia otsakkeita. (Pollard 2019, luku 1.)

4.4.2 Metodit

HTTP-kyselyn metodilla voidaan määritellä, mitä kyselyllä halutaan tehdä. Esimerkiksi tavallinen verkkosivun lataus hoidetaan GET-metodilla, mutta taas esimerkiksi käyttäjän luonti tapahtuu POST-metodilla. Nämä kaksi ovat myös yleisimmin käytettyjä metodeja. Muita yleisiä metodeja ovat PUT ja DELETE. Nämä neljä mainittua metodia muodostavat yhdessä mahdollisuuden CRUD-toiminnallisuudelle, eli sovelluksella pystyy tällöin luomaan, hakemaan, muokkaamaan ja poistamaan resursseja. (HTTP Methods n.d.)

GET-metodia tulisi käyttää lähinnä informaation tai resurssin hakemista varten. Kyselyn-URI:ssa määritellään haettavan resurssin tunniste. Sitä käyttävän kyselyn ei pitäisi pystyä muokkaamaan palvelimella olevaa resurssia, vaan vain ja ainoastaan lukemaan sitä. Peräkkäisillä GET-kyselyillä samoilla parametreilla pitäisikin tulla joka kerta sama vastaus. (RFC 2616:1999, 50-52.)

POST-metodia on HTTP spesifikaation mukaan tarkoitettu käytettävän lähinnä uusien resurssien luontiin. POST-metodia käyttäen voidaan esimerkiksi lähettää täytetty lomake palvelimelle. (RFC 2616:1999, 53-54.)

PUT-metodia käytetään olemassa olevan resurssin päivittämiseen. Teknisesti se toimii samalla tavalla kuin POST, mutta niiden välinen ero on siinä, että POST-metodilla tehdään esimerkiksi uusi käyttäjä tietokannan käyttäjätauluun, mutta PUT-metodilla tiedetään muokattavan käyttäjän tunniste, jolloin muokataan käyttäjätaulussa olevaa käyttäjää. Tällöin siis PUT-metodi tarvitsee muokattavan resurssin tunnisteen kyselyn URI:in (HTTP Methods n.d.)

DELETE-metodia käytetään nimensä mukaisesti resurssin poistoa varten. Kyselyn URI:ssa määritellään poistettavan resurssin id. (RFC 2616:1999, 55.)

4.5 REST

Lyhenne sanoista Representational State Transfer, REST ei itsessään ole käytettävä teknologia, vaan arkkitehtuurityyppi rajapinnoille. Jotta rajapinnan voitaisiin sanoa olevan RESTful, sen täytyy täyttää seuraavat kuusi vaatimusta (What is REST n.d.):

- Asiakas (eng. client) – palvelin (eng. server) -rakenne
- Tilaton (eng. stateless). Jokaisen palvelimelle saapuvan kyselyn täytyy sisältää kaikki tarvittava kyselyn käsittelyä varten.
- Kyselyn vastaukseen täytyy määritellä, voidaanko se tallentaa välimuistiin ja näin käyttää myöhemmissä vastaavissa kyselyissä.
- Rajapinnat täytyy suunnitella yhtenäisiksi ja selkeiksi. Tämä auttaa myös pitämään koko järjestelmäarkkitehtuuria selkeämpänä
- Asiakasohjelman (client) ei tarvitse tietää mihin sovelluksen osaan se tekee kyselyjä, eli tekeekö se kyselyjä esimerkiksi välityspalvelimelle vai suoraan oikealle palvelimelle
- Rajapinnan kautta voi myös siirtää ohjelmakoodia (Vapaaehtoinen).

RESTille ominaista on, että rajapintojen kautta kysytään jotain resurssia. Resurssi voi olla mikä tahansa asia, esimerkiksi ihminen tai koira, joka voidaan identifioida id:llä. Rajapinnan kautta voidaan kysyä resurssin nykyistä tilaa, lisätä uusia sekä muokata tai poistaa olemassa olevia. HTTP-metodit sopivat hyvin näihin käyttötarkoituksiin, mutta niitä ei ole kuitenkaan pakko käyttää, kuten ei myöskään koko HTTP-protokollaa, koska sitä ei ole RESTin vaatimuksissa. (What is REST n.d.)

5 Kehitystyö

5.1 Lähtötilanne

Toimeksiantaja tarvitsi web-käyttöliittymän, josta voitaisiin nähdä toimeksiantajan ylläpitämien palvelujen haavoittuvuustilanne, sekä pystyä tarkastelemaan tarkempia haavoittuvuusskannerin antamia raportteja. Kaikkia toimeksiantajan antamia vaatimuksia ei voida tuoda julkisesti esille, mutta oleellisimpia niistä olivat, että päänäkymässä nähtäisiin tuotteeseen liitettyjen palveluiden kokonaiskuva haavoittuvuuksien osalta. Päänäkymästä pitäisi siis nähdä kaikki testitulokset yleisellä tasolla, esimerkiksi kuinka monta minkäkin vakavuusluokan haavoittuvuutta skanneri on löytänyt mistäkin palvelusta.

Lisäksi käyttöliittymässä tuli pystyä tarkastelemaan palvelukohtaisia haavoittuvuusraportteja, sekä pystyä näkemään myös menneiden skannauksien raportteja erikseen määritellyltä ajalta. Nämä raportit tuli pystyä myös lataamaan käyttäjän työasemalle. Palvelun sisältämän tiedon arkaluontoisuuden vuoksi käyttöliittymän täytyi olla myös käyttöoikeusroolin takana.

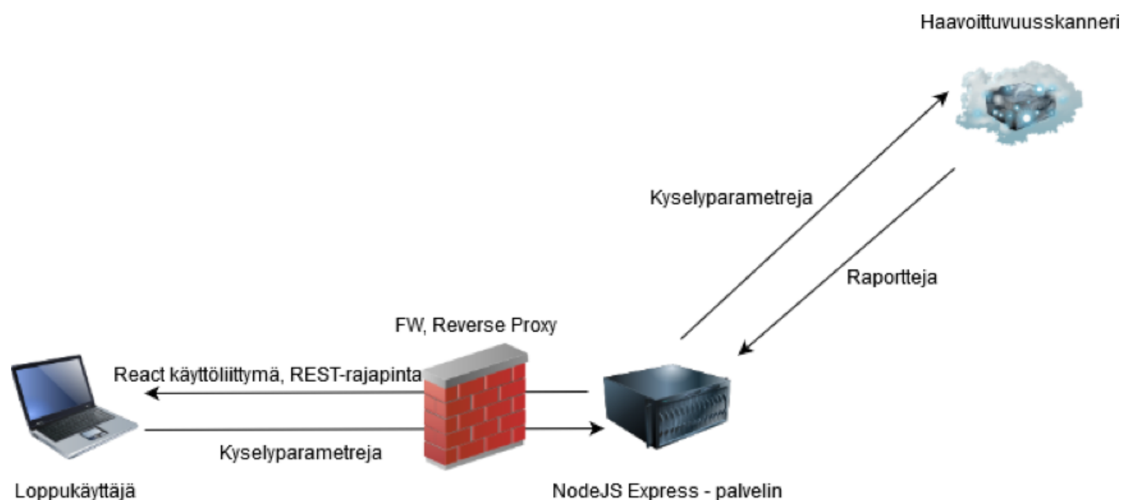
Muihin vaatimuksiin kuului, että käyttöliittymän täytyy toimia ainakin Google Chrome -selaimella, eli esimerkiksi Internet Explorerin rajoituksia ei tarvinnut ottaa kehityksessä huomioon. Käyttöliittymää ei ole myöskään tarkoitus käyttää mobiililaitteilla, joten sitä ei tarvinnut suunnitella responsiiviseksi.

Kun vaatimukset olivat selkeät, voitiin miettiä alustavia teknologiavalintoja. Pienen pohdinnan jälkeen palvelimen teknologiaksi valikoitui Node.js, sekä sen web-sovelluskehityskirjasto Express. Käyttöliittymän täytyy jatkuvasti kysellä palvelimelta lisää tietoa REST-rajapinnan kautta, minkä toteuttamista varten Express soveltuu loistavasti selkeytensä ja yksinkertaisuutensa vuoksi.

Käyttöliittymän teknologiakehykseksi valikoitui React, koska se oli jo ennestään tuttu ja soveltui ominaisuuksiltaan hyvin käyttötarkoitukseen. Reactin ohella kovaan käyttöön tulisi myös JavaScriptin fetch-API, jolla voidaan helposti suorittaa AJAX-kutsuja palvelimelle.

5.2 Suunnittelu

Tarkoituksena oli tehdä yksinkertainen Proof of Concept -tyyppinen ratkaisu, joka täyttäisi aiemmin mainitut vaatimukset. Järjestelmään liittyvät liikkuvat osat olisivat siis haavoittuvuusskanneri rajapintoineen ja NodeJS Express palvelin, joka hakee itsenäisesti tietoa skannerilta, joita sitten tarjoaa palvelimella toimivan REST-rajapinnan avulla käyttöliittymän kautta loppukäyttäjälle. Näillä tiedoilla päästiin yksinkertaiseen Kuvio 7 kuvailtuun järjestelmäarkkitehtuuriin.

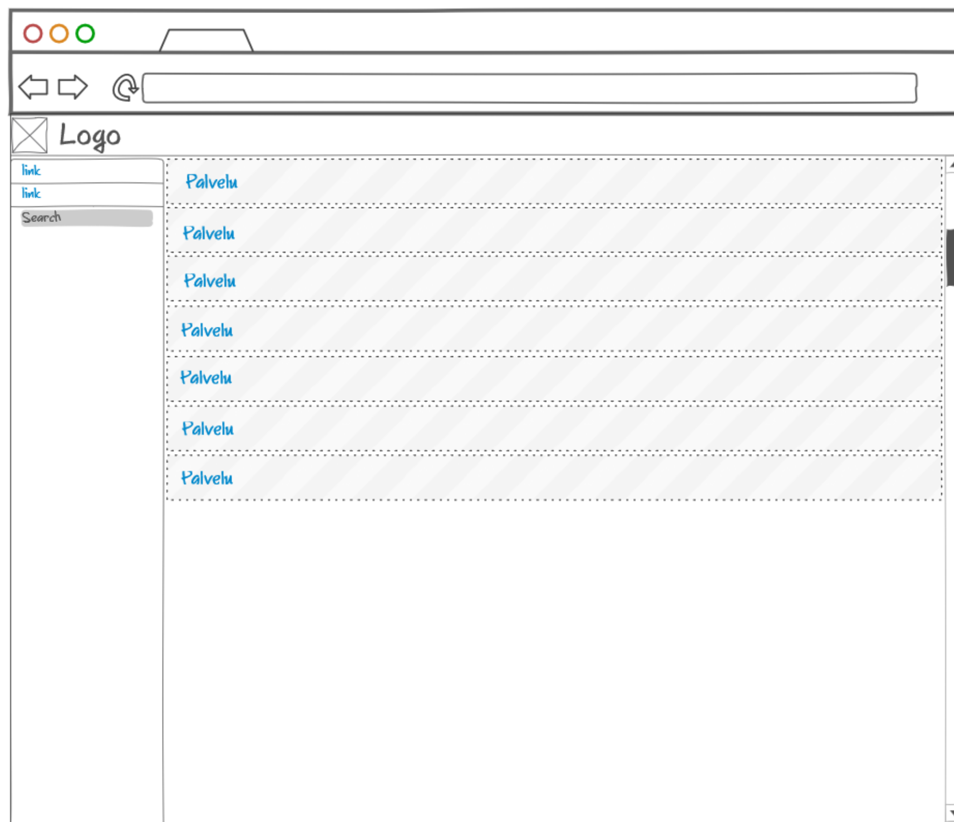


Kuvio 7. Arkkitehtuurikuvaus

Käyttöliittymä suunniteltiin olemaan mahdollisimman yksinkertainen, kuitenkin sen rajoissa, ettei kuitenkaan liikaa karsita näytettävän tiedon määrää. Koko järjestelmän tarkoitus on kuitenkin kerralla nähdä haavoittuvuuksien kokonaiskuva palvelujen kesken, sekä mahdollisuus nähdä jokaisesta palvelusta tarkempi haavoittuvuusraportti. Käyttöliittymästä oli helppo suunnitella yksinkertainen annettujen suoraviivaisten vaatimusten pohjalta.

Sovellukseen suunniteltiin siis kaksi oleellista näkymää: päänäkymä, sekä päänäkymässä palvelua klikattaessa yksittäiselle palvelulle avautuva palvelukohtainen näkymä. Kuvio 8 mukaiseen päänäkymään listataan kaikki järjestelmään liitetyt palvelut. Jokaiselle palvelulle varattuun riviin liitetään näihin suoritettujen haavoittuvuustestauksien tuloksien yleiskatsaukset. Eli lähinnä kuinka monta ja minkä tasoisia haavoittuvuuksia palveluun liittyy.

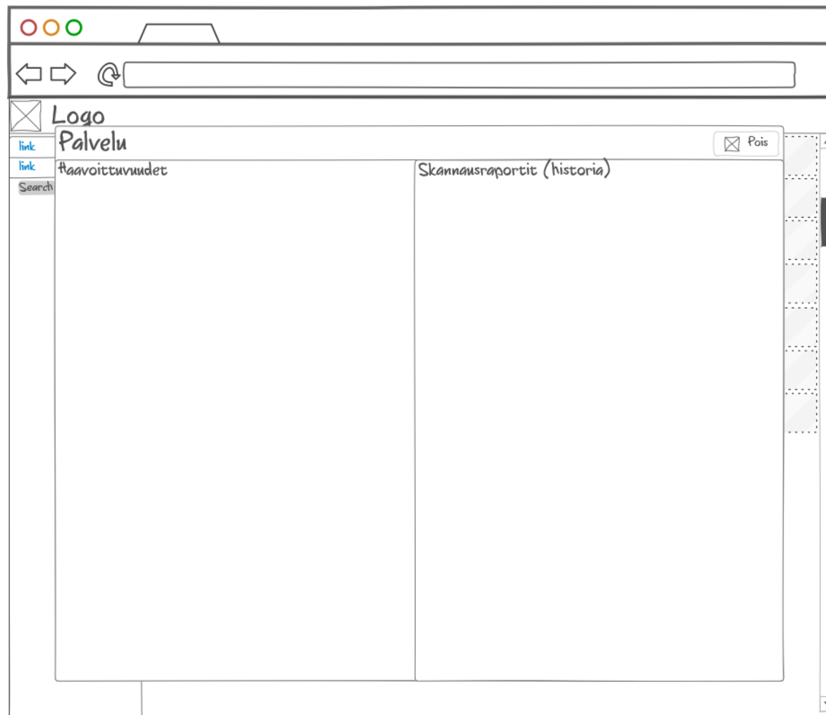
Päänäkymän vasempaan reunaan suunniteltiin navigointipalkki, jonka avulla voidaan liikkua eri palvelulistauksien välillä. Tulevaisuudessa tuotteeseen liitettyjen palveluiden määrä todennäköisesti vain kasvaa, jota varten käyttöliittymään olisi hyvä lisätä myös hakukenttä, jolla voidaan rajata näkymässä esitettyjä palveluita.



Kuvio 8. Käyttöliittymän päänäkymän mockup

Päänäkymässä tulisi näkyä listaus järjestelmään kytketyistä palveluista sekä nopea katsaus niihin suoritettujen testauksien tuloksista. Päänäkymässä kun käyttäjä klikkaa haluamaansa palvelua avattaisiin päänäkymän päälle uuteen ikkunaan Kuvio 9 mu-

kainen näkymä. Tässä palvelukohtaisessa näkymässä halutaan nähdä tarkemmat tiedot kyseisestä palvelusta. Näitä tarkempia tietoja voi olla esimerkiksi skannaushistoria ja aikaisemmat haavoittuvuusraportit, sekä skannauksien kohteet palvelintasolla.



Kuvio 9. Palvelukohtaisen näkymän mockup

5.3 Toteutus

Tarpeelliset palvelimet ja ohjelmistot löytyivät toimeksiantajalta valmiina, joten itse toteutus päästiin aloittamaan nopealla aikataululla. Toteutuksen kehitysympäristönä käytettiin Visual Studio Code -tekstieditoria pääasiassa Ubuntu-pohjaisella työasemalla. Tuotetta kehitettiin mukaillen ketteriä menetelmiä, joten tuotteeseen liittyvät vaatimukset ja suunniteltu ulkoasu saattoivat kehityksen aikana muuttua.

Toteutusta lähdettiin toteuttamaan MVP-tyyppisenä kokonaisuutena, eli tarkoitus oli saada mahdollisimman vähällä työllä toimiva kokonaisuus. Ensimmäisenä syntyi Node.js Express-palvelin. Tämän tehtävänä on hakea itsenäisesti tietyllä aikavälillä

uusimmat tiedot haavoittuvuusraportilta. Palvelimelle oli myös toteutettava rajapinta, jonka kautta voitiin hakea haavoittuvuusraporttien tuloksia skanneriohjelmistolta tämän REST-rajapinnan yli.

Kuvio 10 on kuvattuna esimerkki palvelimen suorittamasta kyselystä haavoittuvuusskannerin rajapintaan kirjoitettuna Node.js:n Express-kirjastolla. Kutsuttaessa selaimella palvelimen IP-osoitetta reitistä */api/test/*, palvelin lähettää heti ennalta määritetyn kyselyn fetch-kirjaston avulla haavoittuvuusskannerille, minkä jälkeen palauttaa vastauksen käyttäjälle selaimeen käyttämällä *res.send()* -funktiota. Tällä tavoin toteutettuna käyttäjä ei suoraan pääse tekemään haavoittuvuusskannerille kyselyitä. Kun haavoittuvuusskannerille voidaan lähettää kyselyitä vain palvelimen kautta, sovelluksen väärinkäyttöriski pienenee, koska skannerille lähetetään vain palvelimelle koodattuja kyselyitä, joka pienentää mahdollista hyökkäyspinta-alaa merkittävästi.

```
app.get('/api/test', async (req, res) => {
  const response = await fetch(`[illegible]`, {
    method: 'GET',
    mode: 'cors',
    headers: {
      "Content-Type":"application/json",
      [illegible]
    }
  })

  const parsed = await response.json()

  res.setHeader('Content-Type', 'application/json');
  res.send(JSON.stringify(parsed))
});
```

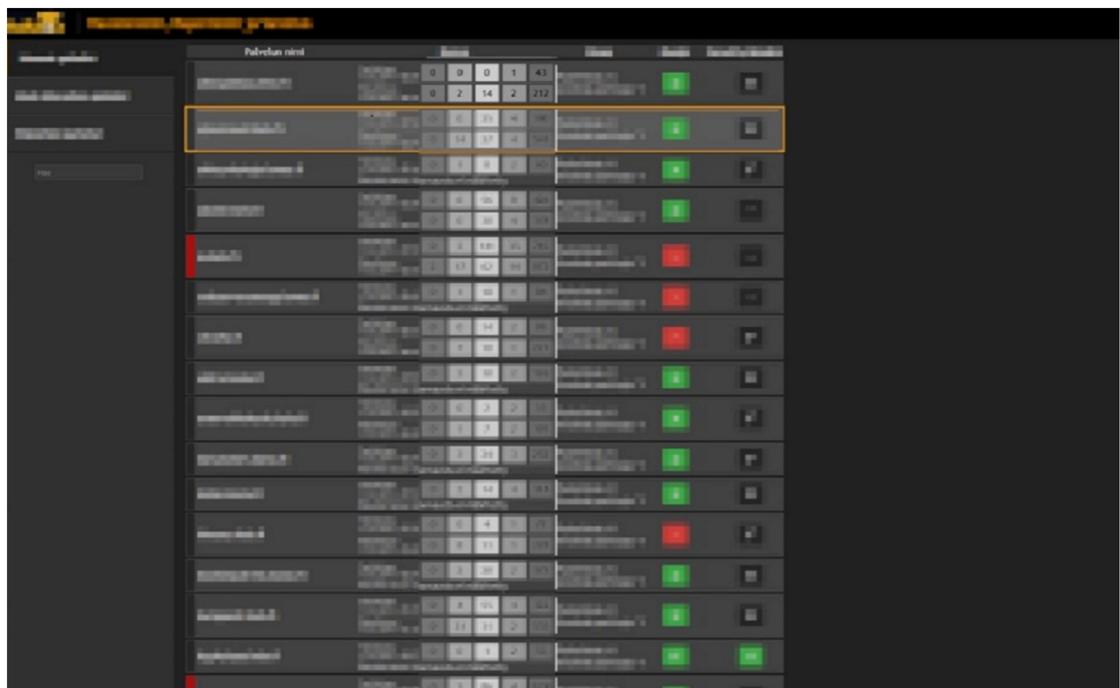
Kuvio 10. Esimerkki palvelinpuolen HTTP-kyselystä

Palvelimen ohelle ja sen ylläpidettäväksi syntyi myös React-pohjainen käyttöliittymä. Käyttöliittymää kehitettiin ketterästi toimeksiantajan toiveita kuunnellen, sekä työn aikana ilmaantuneet ideat huomioon ottaen. Käyttöliittymää kehitettäessä pääasiassa otettiin huomioon uusimpia hyviä käytänteitä, esimerkiksi luokkapohjaisia React-komponentteja ei luotu ollenkaan, vaan kaikissa tilallisissa komponenteissa käytettiin React Hookkeja. Toteutuksessa käytettiin JavaScriptin uusia ECMAScript-standardin ominaisuuksia.

5.4 Tulokset

Lopputuotoksena syntyi web-sovellus, joka vastasi annettuja vaatimuksia. Lopputuotteessa on kolme erillistä näkymää: päänäkymä, palvelukohtainen näkymä, sekä raporttinäkymä. Lopputuotosta havainnollistavista kuvioista on osa kohdista muutettu mustavalkoisiksi sekä suurin osa tekstistä sumennettu toimeksiantajan toiveesta.

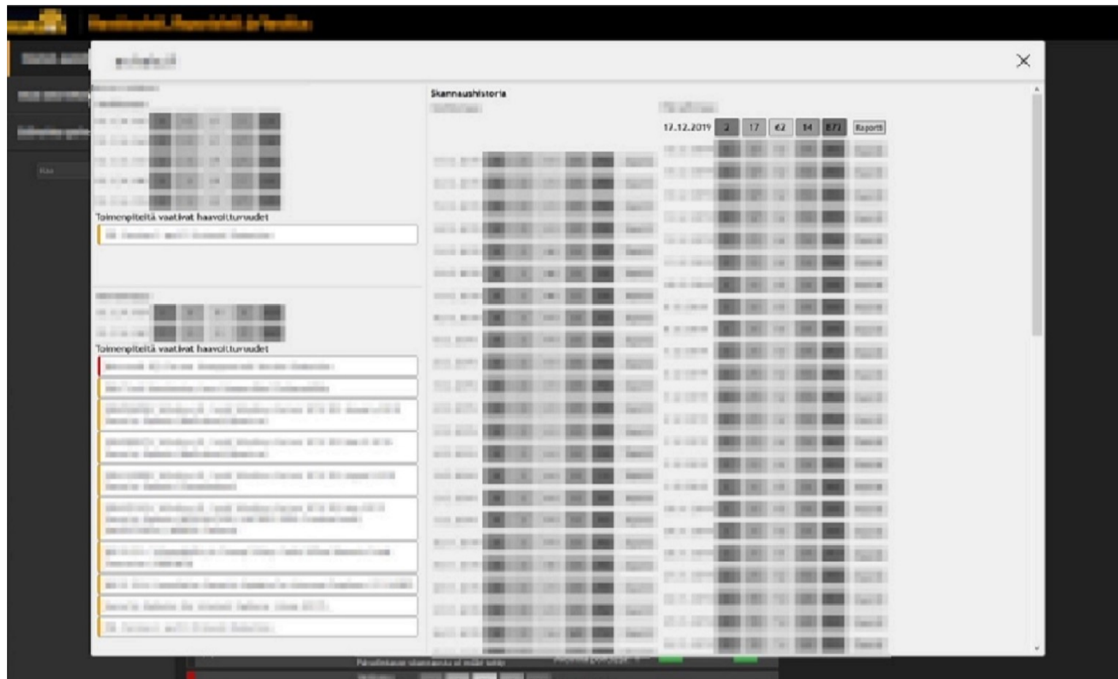
Kuvio 11 mukainen päänäkymä listaa jokaisen järjestelmään määritetyn palvelun. Sen lisäksi palveluista näkyvät myös skannaustulokset yhteenlaskettuna sekä skannausaika. Näkymään lisättiin myös hakukenttä, jotta käyttäjän ei tarvitse etsiä haluamaansa palvelua pitkästä listasta. Minkä tahansa palvelun kohdalta klikkaamalla avautuu kyseiseen palveluun liittyvä näkymä.



Kuvio 11. Palvelulistausnäkymä

Palvelukohtainen näkymä luotiin näyttämään tarkempaa tietoa halutusta palvelusta. Kuvio 12 mukainen näkymä sisältää palvelun tarkat tiedot, kuten palvelinkohtaiset haavoittuvuudet sekä skannaushistorian. Näkymän vasemmalla puollella on listattu yleiskatsaus skannatuista kohteista ja vakavimpia palvelussa esiintyviä haavoittu-

vuuksia on nostettu suoraan näkyville. Tällaista klikkaamalla käyttäjälle avautuu verkkosivu, jolla lukee tarkempaa tietoa kyseisen haavoittuvuuden aiheuttajasta ja korjausmahdollisuuksista.

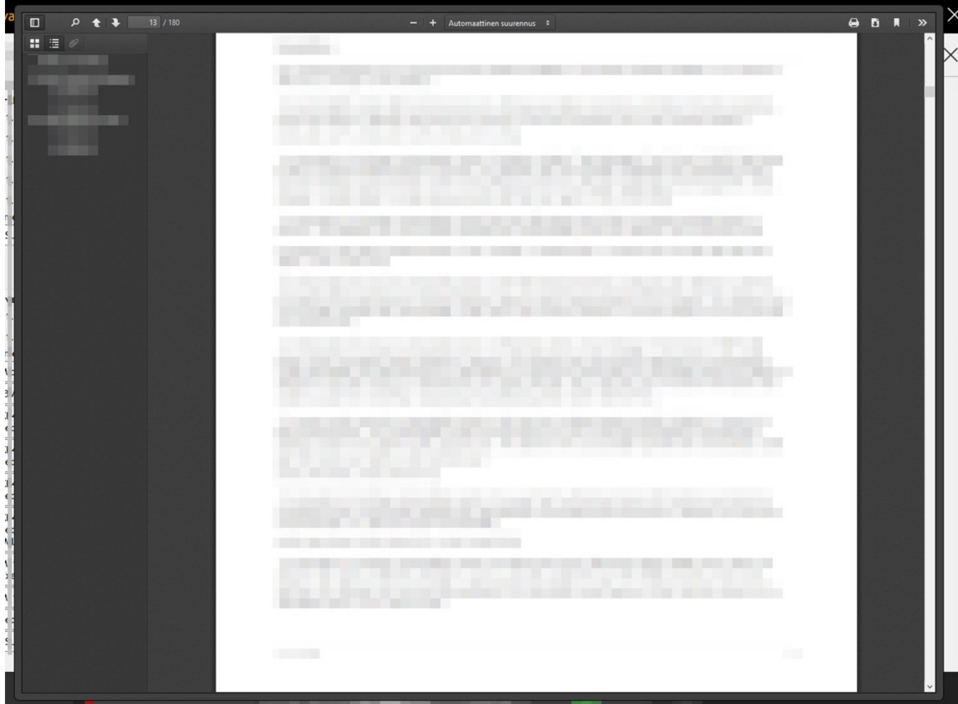


Kuvio 12. Palvelukohtainen näkymä

Näkymän oikea puoli on varattu menneille skannauksille noin kuukauden ajalta, jotta voitaisiin nähdä, missä vaiheessa uusi haavoittuvuus on järjestelmään ilmaantunut. Jokaisen skannauspäivämäärän ja tuloksien oikealla puolella on ”Raportti”-nappula, jota painamalla sovellus hakee palvelimelta PDF-raportin kyseisestä skannauksesta ja avaa sen uutena ikkunana käyttöliittymän päälle.

Raporttinäkymään itse raportti tulee base64-muodossa, joka asetetaan iframe-elementin lähteeksi. Tämä toteutus toimii testatusti vain Chrome- ja Firefox-selaimissa. Microsoft Edgellä PDF-tiedostoa ei voi avata iframe-elementtiin vastaavalla tavalla, minkä takia raporttinäkymä ei aukea lainkaan, vaan raportti latautuu suoraan tietokoneelle PDF-tiedostona.

Kuvio 13 esitelty näkymä on Firefoxilla avattu, Chromella näkymä on hieman erilainen johtuen siitä, että näkymässä käytetään selaimen omaa PDF-lukijaa. Tästä näkymästä löytyy kuitenkin selaimesta riippuen myös ohjausnappulat PDF-tiedoston lataamiselle omalle työasemalle.



Kuvio 13. Raporttinäkymä

Loppuvaiheessa tuotteen koettiin täyttävän annetut vaatimukset, jolloin se jätettin testikäyttöön tuotantopalvelimelle. Tuotetta vasten ei suoritettu virallisia testitapauksia, vaan testaus hoidettiin kehityksen ohessa tiimin sisällä käyttämällä työkalua muun työn tukena. Testikäytössä sovellukselle saatiin selkeämpää suuntaa esimerkiksi raporttien muodon osalta, sekä esimerkiksi esille nousi tarve viimeisimmän skannauksen tarkasta ajankohdasta sovelluksen etusivulle, jolloin se lisättiin näkyville. Ohjelmiston tulevaisuuden osalta alettiin valmistelemaan virallista käyttöönottoa, joka voi toimia myös osaltaan mittarina opinnäytetyön onnistumiselle.

6 Pohdinta

Työn tavoitteena oli luoda toimeksiantajalle, eli Organisaatio X:lle, web-käyttöliittymä haavoittuvuusskannerin tuloksien raportointia varten. Kehitystyötä täydennettiin myös tutkimalla tarkemmin haavoittuvuusskannerien käyttökohteita sovelluksen elinkaaren aikana.

Työn lopputuloksena syntyi vaatimuksensa täyttävä, eli pääpiirteittäin varsin onnistunut web-sovellus. Sovelluksen tarpeen taustatietoa varten tehtiin tutkimuskysymys ”Kuinka haavoittuvuusskannereita voidaan hyödyntää sovelluksen elinkaaren aikana sen tietoturvallisuuden parantamiseksi?”, johon vastattiin kehittämistutkimukseen sisältyvän laadullisen tutkimuksen, eli käytännössä teoriaosuuden avulla hyvin yleisellä tasolla teoriaosiossa tarkasteltujen haavoittuvuusskannerien näkökulmasta.

Kehityksen aikana skannereita on hyvä käyttää säännöllisesti automatisoituna esimerkiksi kehittäjän versionhallintaan tekemän muutoksen seurauksena tai vaihtoehtoisesti tasaisen ajan välein. Tuotannossa olevien sovelluksien teknologiaratkaisut vanhenevat koko ajan ja niistä löydetään jatkuvasti uusia haavoittuvuuksia. Säännöllisillä skannauksilla näitä haavoittuvuuksia voidaan löytää ajoissa ja pystyä suorittamaan mahdolliset korjaustoimenpiteet. Organisaatioissa, joissa ylläpidettäviä sovelluksia on paljon, voi auttaa kustomoitu käyttöliittymä, johon voidaan hakea halutut raportit selkeään muotoon, kuten työn lopputuotteessa tehtiin.

Tuotetun sovelluksen teknologiaratkaisut palvelivat käyttötarkoitusta hyvin. React sopi ominaisuuksiltaan interaktiivisen käyttöliittymän taustateknologiaksi. Palvelimen ei tarvinnut olla kovin monimutkainen, joten sen yksinkertaisen REST-rajapinnan käsittely voitiin hoitaa Express-kirjaston avulla ongelmitta. Kummatkin teknologiavalinnat olivat jo ennestään tuttuja, joten niiden opetteluun ei kulunut turhaan aikaa. Käyttöliittymään haettiin raportit yhdeltä haavoittuvuusskannerilta.

Sovellus täyttää määritellyt vaatimukset ja käyttötapaukset. Sillä nähdään listattujen palveluiden haavoittuvuuksien kokonaiskuva nopeasti, voidaan tarkastella skannaus-

historiaa, sekä lukea haavoittuvuusraportteja. Ei-toiminnallisia vaatimuksia sovellukselle ei juurikaan määritelty. Osaltaan tämä oli myös kehitystä hankaloittava seikka, koska toimeksiantajalta saadut vaatimukset olivat melko vähäisiä. Jälkeenpäin ajateltuna sovelluksen suunnitteluun olisi voinut käyttää enemmän aikaa nimenomaan vaatimusten osalta, jolloin itse kehitys olisi ollut suoraviivaisempaa. Myös kehitys olisi voinut olla hieman järjestelmällisempää, esimerkiksi enemmän kanban-tyyliä noudattaen. Yhden henkilön kehittämänä kuitenkin ketterien menetelmien kaavojen noudattaminen täydellisesti on melko tarpeetonta.

Valmis lopputuotos jätettiin toimeksiantajalle käyttöön ja sitä varten kirjoitettiin käyttö- ja jatkokehitysohjeet. Tämä toimii myös hyvänä mittarina opinnäytetyön onnistumisesta. Työn lopputulosta voidaankin sanoa tältä pohjalta onnistuneeksi, eikä kehityksessäkään ilmennyt kuitenkaan suurempia ongelmia. Suurimmat haasteet johtuivat toimeksiantajan monimutkaisista tuotantoonviemisprosesseista.

Sovelluksen jatkokehitykselle oli ideoita ja näkemyksiä, joista kaikkia ei voida tuoda julkisesti esille kovin tarkalla tasolla. Käyttöliittymään jätettiin tilaa esimerkiksi muille mahdollisille skannauksille tai testauksille, jolloin voitaisiin saada laajempaa näkemystä palveluiden tietoturvallisuuden tilasta. Yksi mahdollinen lisäys käyttöliittymään voisi olla mahdollisuus kirjata haavoittuvuudelle tehtyjä toimenpiteitä, jolloin näkisi selkeästi, onko kyseiselle haavoittuvuudelle jo tehty jotain ja jos on niin mitä. Tällaisen toiminnan lisäämiseksi tarvittaisiin yksinkertainen tietokantarakenne, jota nykyisessä toteutuksessa ei ollut.

Lähteet

2016 Cost of Data Breach Study: Global Analysis. 2016. IBM:n julkaisema raportti. Viitattu 13.4.2020. https://www.ibm.com/downloads/cas/7VMK5DV6.

About CVE. 2019. CVE:n viralliset sivut. Viitattu 4.12.2019. https://cve.mitre.org/about/index.html.

About The Open Web Application Security Project. 2019. OWASP viralliset sivut. Muokattu 18.11.2019. Viitattu 11.12.2019. https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project.

Building a DevSecOps Culture - from a Technical Perspective. N.d. Tech guide from the USA government. https://tech.gsa.gov/guides/building_devsecops_culture/.

Chacon, S & Straub, B. 2014. Pro Git. 2.p. Apress.

Common Vulnerability Scoring System v3.1: Specification Document. N.d. CVSS:n määrittelydokumentti. Viitattu 4.4.2020. https://www.first.org/cvss/v3.1/specification-document.

Copes, F. 2018. The Node.js Handbook. Kirjoittajan itse julkaisema e-kirja. Viitattu 1.4.2020. https://flaviocopes.nyc3.digitaloceanspaces.com/node-handbook/node-handbook.pdf.

Data Breach 101: Top 5 Reasons it Happens. N.d. Whoa, kyberturva-alan yritys. Viitattu 13.4.2020. https://www.whoa.com/data-breach-101-top-5-reasons-it-happens/.

General Information. N.d. NIST. NVD:n yleinen informaatiosivu. Viitattu 22.3.2020. https://nvd.nist.gov/general.

Get Started. 2020. Tenable. Tenable.io:n sovellusrajapintojen dokumentaatio. Viitattu 2.4.2020. https://developer.tenable.com/docs.

Introducing Hooks. 2020. Reactin virallinen dokumentaatio. Viitattu 5.4.2020. https://reactjs.org/docs/hooks-intro.html.

Introducing JSX. 2020. Reactin virallinen dokumentaatio. Viitattu 5.4.2020. https://reactjs.org/docs/introducing-jsx.html.

JavaScript. 2012. Mozillan dokumentti kehittäjille. Muokattu 30.3.2020. Viitattu 4.4.2020. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

HTTP Methods. N.d. Restfulapis dokumentaatio HTTP-metodien käytölle. Viitattu 5.4.2020. https://restfulapi.net/http-methods/.

Huotarinen, J, 2016. Tiesitkö jatkuvan julkaisun olevan jo arkipäivää?. Gofore ohjelmistoyrityksen blogikirjoitus. Julkaistu 20.4.2016. Viitattu 5.4.2020. https://gofore.com/tiesitko-jatkuvan-julkaisun-olevan-jo-arkipaivaa/.

Kananen, J. 2015. Kehittämistutkimuksen käytännön opas: Miten kirjoitan kehittämistutkimuksen vaihe vaiheelta. Jyväskylän ammattikorkeakoulun julkaisuja 212. Viitattu 22.3.2020. https://janet.finna.fi/. Booky.fi.

Klemetti, M. 2013. Mitä on devops?. Eficoden julkaisu. Viitattu 5.4.2020. https://www.eficode.com/blogi/blogi/mita-on-devops.

OWASP Application Security Verification Standard. N.d. ASVS projektin kotisivu. Viitattu 21.3.2020. https://owasp.org/www-project-application-security-verification-standard/.

OWASP Top 10 – 2017: The Ten Most Critical Web Application Security Risks. 2017. OWASPin julkaisu. Viitattu 16.9.2019. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

OWASP ZAP. N.d. Zed Attack Proxy:n projektisivu. Viitattu 16.9.2019. https://github.com/zaproxy/zaproxy.

Pollard, B. 2019. HTTP/2 in Action. Manning Publications.

React. 2019. React-kirjaston virallinen sivu. Viitattu 4.12.2019. https://reactjs.org/.

React.Component. 2020. Reactin kehittäjädokumentaation osa. Viitattu 4.4.2020. https://reactjs.org/docs/react-component.html.

RFC 2616:1999, Hypertext Transfer Protocol -- HTTP/1.1 -spesifikaatio. Viitattu 18.12.2019. https://tools.ietf.org/html/rfc2616.

RFC 7540:2015, Hypertext Transfer Protocol Version 2 (HTTP/2) -spesifikaatio. Viitattu 18.12.2019. https://tools.ietf.org/html/rfc7540.

Source Code Analysis Tools. N.d. OWASPIN listaus koodianalyysityökaluille. Viitattu 5.4.2020. https://owasp.org/www-community/Source_Code_Analysis_Tools.

The Burp Suite Family. 2020. PortSwigger. Burp Suiten kotisivu. Viitattu 21.3.2020. https://portswigger.net/burp.

Terminology. 2017. CVE. CVE:n määritelmät. Viitattu 22.3.2020. Muokattu 15.12.2017. https://cve.mitre.org/about/terminology.html.

The Nessus Family. 2019. Tenable. Nessuksen kotisivu. Viitattu 4.12.2019. https://www.tenable.com/products/nessus.

Vulnerabilities. N.d.a. NIST. Viitattu 4.12.2019. https://nvd.nist.gov/vuln.

Vulnerabilities. N.d.b. OWASPin määritelmä. Viitattu 22.3.2020. https://owasp.org/www-community/vulnerabilities/.

Vulnerabilities By Type. N.d. Taulukko CVE Details -sivustolla. Viitattu 12.12.2019. https://www.cvedetails.com/vulnerabilities-by-types.php

Vulnerability Metrics. N.d. NIST. Viitattu 4.12.2019. https://nvd.nist.gov/vuln-metrics/cvss.

What is Continuous Integration?. N.d. Amazon. Artikkeli Amazonin AWS:n tuotesivulla. Viitattu 22.3.2020. https://aws.amazon.com/devops/continuous-integration/.

What is DevSecOps. N.d. Artikkeli Red Hatin sivuilla. Viitattu 21.3.2020. https://www.redhat.com/en/topics/devops/what-is-devsecops.

What is REST. N.d. REST-rajapintojen määritelmä. Viitattu 22.3.2020. https://restfulapi.net/.

Zalewski, M. 2011. Tangled Web: A Guide to Securing Modern Web Applications. No Starch Press.

ZAP API Documentation. N.d. Official OWASP ZAP api documentation. https://www.zaproxy.org/docs/api/.