Maisterintutkielma

Tietojenkäsittelytiede

Web-sovellusten tietoturvan automaattinen regressiotestaus

Tomi Simsiö

27.11.2020

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Ohjaaja

Prof. Valtteri Niemi

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytiede

Tekijä — Författare — Author

Tomi Simsiö

Työn nimi — Arbetets titel — Title

Web-sovellusten tietoturvan automaattinen regressiotestaus

Ohjaajat — Handledare — Supervisors

Prof. Valtteri Niemi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Maisterintutkielma	27.11.2020	45 sivua

Tiivistelmä — Referat — Abstract

Web-sovellusten tietoturva on jokaiselle internetin käyttäjälle tärkeä asia. Yksittäiseen sivustoon tai web-palveluun tapahtuvasta tietomurrosta voi seurata palvelun käyttäjille paljon harmia, joka olisi usein ollut estettävissä. Pienellä vaivalla toteutettavissa olevalla automaattisella tietoturvaskannauksella pystytään havaitsemaan suuri osa web-sovellusten haavoittuvuuksista.

Tässä työssä tutkitaan automaattisen tietoturvaskannauksen liittämistä sovelluksen kehitysprosessiin. Lisäämällä tietoturvaskannaus osaksi kehitettävien sovelluksien jatkuvaa koontiajoa pyritään parantamaan sovellusten tietoturvaa havaitsemalla uudet haavoittuvuudet heti niiden synnyttyä. Automaattisen tietoturvaskannauksen käyttämisellä osana regressiotestausta pyritään myös vähentämään käsin tehtävän testauksen määrää, sekä parantamaan sovelluksen tietoturvaa kehitysprosessin jokaisessa vaiheessa.

Tämän työn tavoitteena on tuottaa yksinkertainen ja tehokas GitLab CI -koontiajoon liitettävä tietoturvaskannauksen tekevä komponentti, joka on helppo lisätä myös tuleviin sovellusprojekteihin. Komponentti hyödyntää Arachni- ja ZAP-nimisiä avoimen lähdekoodin skannausohjelmistoja.

Työssä selvitetään samalla, kuinka helppoa tämä tietoturvatarkistuksen tekevä komponentti on pystyttää ja miten pienellä vaivalla sen uudelleenkäyttö muissa sovellusprojekteissa onnistuu. Lisäksi tutkitaan, kuinka tehokkaasti toteutettu järjestelmä löytää haavoittuvuuksia sovelluksista.

ACM Computing Classification System (CCS)
Security and privacy → Software and application security → Web application security
Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Avainsanat — Nyckelord — Keywords

Tietoturva, tietoturvatestaus, regressiotestaus, web-sovellus, Arachni, ZAP

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Ohjelmistojärjestelmien erikoistumislinja

Sisällys

1 Johdanto

Internet ja web-sovellukset ovat osa lähes jokaisen arkipäivää. Eurooppalaisista 86% käyttää internetiä päivittäin [Eur19]. Useat verkkopalvelut käsittelevät henkilötietoja ja muuta arkaluontoista materiaalia. Etenkin näiden palveluiden kohdalla tietoturvasta huolehtiminen on ensisijaisen tärkeää.

Open Web Application Security Project (OWASP) -organisaation Top 10 -projekti [OWA17] ylläpitää listaa kymmenestä yleisimmästä web-sovellusten riskityypistä. Projektin tavoitteena on lisätä tietoisuutta web-sovellusten tietoturvasta ja siitä on muotoutunut yleisesti tunnettu ja useasti viitattu dokumentti. Se toimii hyvänä perustana ja muistilistana web-sovellusten tietoturvatestauksessa. Projektin luokittelun pohjalta on myös käytännöllistä vertailla *penetraatiotestauksessa* (penetration testing) käytettävien ohjelmistojen ominaisuuksia tai toiminnallisuutta.

Penetraatiotestaus on yleinen tapa testata sovelluksen tietoturvaa. Siinä testaaja lähestyy sovellusta käyttäjän näkökulmasta, tietämättä mitään testattavan sovelluksen sisäisestä toimintalogiikasta. Testaaja yrittää murtautua sovellukseen samoin keinoin kuin potentiaalinen hyökkääjä yrittäisi ja pyrkii löytämään sovelluksessa olevat haavoittuvuudet [MM14, s. 14].

Haavoittuvuudet, eli tietoturva-aukot ovat sovelluksessa olevia virheitä, joiden avulla hyökkääjä pystyy tekemään sovelluksessa toimenpiteitä, joita hänen ei pitäisi päästä tekemään.

Web-sovellusten automaattiseen penetraatiotestaukseen on saatavilla runsaasti sekä kaupallisia, että ilmaisia ohjelmistoja. Ilmaisista ohjelmistoista suuri osa on lisäksi julkaistu avoimen lähdekoodin lisensseillä. Tunnettuja ja laajasti käytettyjä sovelluksia ovat esimerkkejä Burp Suite [Por20], Zed Attack Proxy [Ben20b] ja Arachni [Las20a]. Osa ohjelmistoista toimii apuvälineenä manuaalisessa testauksessa, mutta osa kykenee myös täysin itsenäisesti löytämään ja tunnistamaan suuren osan kohdeohjelmiston mahdollisista haavoittuvuuksista. Testaajan manuaalisesti tekemällä penetraatiotestauksella on mahdollista löytää haavoittuvuuksia, joihin automaattisesti ajettu *skannaus* (scanning) ei kykene. Automaattista skannausta kannattaa silti mahdollisuuksien mukaan hyödyntää, sillä sen avulla on mahdollista helposti, halvalla ja pienellä vaivalla löytää suuri osa haavoittuvuuksista.

Esimerkiksi muutamia vuosia sitten hakkeriryhmittymät Anonymous ja LulzSec aiheuttivat paljon harmia ja vahinkoa useille sivustoille ja niiden käyttäjille. Näiden ryhmien tekemät tietovuotoihin johtaneet hyökkäykset olivat keskimäärin hyvin tunnettuja, kuten SQL-injektioita ja *etätiedoston liittämishyökkäyksiä* (remote file inclusion) [Man11]. Hyökkäyksiin johtaneista haavoittuvuuksista suuren osan olisi voinut välttää automaattisilla tietoturvaskannauksilla.

Regressiotestaus (regression testing) on sovellukseen tehdyn muutoksen jälkeistä rutiininomaista testausta. Sillä pyritään varmistamaan, että sovellukseen tehdyn muutoksen jälkeen sovellus toimii edelleen määrittelyn mukaisesti ja ettei muutoksen yhteydessä tapahtunut muita tahattomia muutoksia [DRP99, s. 4].

Tässä työssä tutkitaan automaattisen tietoturvaskannauksen liittämistä sovelluksen kehitysprosessiin osaksi regressiotestausta. Lisäämällä tietoturvaskannaus osaksi kehitettävien sovelluksien jatkuvaa koontiajoa pyritään parantamaan sovellusten tietoturvaa havaitsemalla uudet haavoittuvuudet heti niiden synnyttyä. Tällöin ongelmat saadaan korjattua ennen kuin haavoittuvainen versio päätyy tuotantokäyttöön ja loppukäyttäjien tietoturva ei pääse missään vaiheessa vaarantumaan. Automaattisen tietoturvaskannauksen käyttämisellä osana regressiotestausta pyritään myös vähentämään käsin tehtävän testauksen määrää, sekä parantamaan sovelluksen tietoturvaa kehitysprosessin jokaisessa vaiheessa.

Tutkimusympäristönä on Oikeat Oliot Oy:n [Oli20] GitLab CI/CD -koontipalvelin [Git20e], sekä sovelluskohtaiset testipalvelimet. GitLab CI/CD -palvelin kääntää sekä paketoi kunkin sovelluksen jokaisen versiohallintaan viedyn muutoksen jälkeen ja käynnistää ohjelman testipalvelimelle.

Tämän työn tavoitteena on tuottaa yksinkertainen ja tehokas GitLab CI/CD -koontiajoon liitettävä tietoturvaskannauksen tekevä komponentti, joka on helppo lisätä myös tuleviin sovellusprojekteihin. Komponentti hyödyntää avoimen lähdekoodin skannausohjelmistoa, joka valitaan aikaisemmin julkaistujen suorituskykytestien perusteella.

Työssä selvitetään samalla, kuinka helppoa tämä tietoturvatarkistuksen tekevä komponentti on pystyttää ja miten pienellä vaivalla sen uudelleenkäyttö muissa sovellusprojekteissa onnistuu. Lisäksi tutkitaan, kuinka tehokkaasti toteutettu järjestelmä löytää haavoittuvuuksia sovelluksista.

Luvussa 2 käydään läpi oleellista käsitteistöä. Luvussa 3 esitellään web-sovellusten tietoturvaa ja sen ongelmia yleisesti, sekä OWASP Top 10-listan avulla. Luvussa 4 keskitytään web-sovellusten tietoturvan testaukseen, tarkastellaan avoimen lähdekoodin tietoturvas-

kannausohjelmistojen suorituskykytestejä, valitaan kaksi skannausohjelmistoa tässä työssä käytettäväksi ja esitellään ne tarkemmin. Luku 5 esittelee yrityksen sovellusten koonti- ja testausympäristöt, käy läpi kuinka skannaussovellukset liitetään osaksi koontiprosessia, sekä arvioidaan tehdyn toteutuksen tehokkuuta ja hyödyllisyyttä. Luvussa 6 käsitellään tutkimuksen tulokset ja johtopäätökset. Luvussa 7 annetaan yhteenveto tutkimuksesta.

2 Käsitteistö ja historiaa

Tässä luvussa käydään läpi web-sovellusten, sovelluskehityksen ja tietoturvan peruskäsitteitä, joita tarvitaan jatkossa.

2.1 Jatkuva integraatio

Kun sovelluksen parissa työskentelee useampi kehittäjä, täytyy kunkin kehittäjän tekemä työ ennemmin tai myöhemmin yhdistää yhdeksi toimivaksi sovellukseksi. Mikäli tätä integraatiovaihetta lykätään, kasautuu siitä aiheutuva työ todennäköisesti työlääksi prosessivaiheeksi.

Jatkuva integraatio (Continuous Integration, CI) -periaatteen ideana on, että jokainen sovelluksen parissa työskentelevä kehittäjä integroi työnsä yhteiseen versionhallintaan mahdollisimman usein, yleensä päivittäin. Näin integroinnista aiheutuva työ pilkkoutuu pieniksi heti korjattaviksi tehtäviksi, eikä integraatiotyö pääse muodostumaan omaksi isoksi vaiheekseen sovelluskehitysprosessissa.

Yllä kuvatun CI-prosessin apuna käytetään usein CI-palvelinta, joka valvoo versionhallintaan vietyjä muutoksia ja kääntää sekä suorittaa automaattiset testit jokaisen versionhallintaan viedyn muutoksen jälkeen. Sovellusmuutoksen ja sen integroinnin voidaan katsoa onnistuneen vasta kun CI-palvelin saa käännettyä ja testattua sovelluksen onnistuneesti. CI-palvelimella voi automatisoida myös muita toimintoja, kuten sovelluksen asennuksen testiympäristöön. Suosittuja avoimen lähdekoodin CI-palvelinsovelluksia ovat muun muassa Jenkins ja GitLab CI. [FF06]

2.2 SPA-sovellus

Web-sovellukset ovat selainsovelluksella käytettäviä interaktiivisia verkkosivuja. Palvelimella oleva web-sovellus kommunikoi käyttäjän selaimen kanssa HTTP-protokollalla. Web-sovelluksen palvelimella suoritettava osuus voi olla toteutettu millä tahansa ohjelmointikielellä. Selaimen päässä suoritettavat interaktiiviset toiminnallisuudet täytyy toteuttaa jollain selaimen tukemalla ohjelmointikielellä, yleensä JavaScriptillä. Perinteisissä

web-sovelluksissa palvelimelta haetaan uusi sivu jokaisen käyttäjän tekemän toiminnon jälkeen.

Single page application (SPA) -sovelluksissa web-sivu haetaan palvelimelta selaimeen vain kerran. Käyttäjän navigoidessa sivustolla, siinä tarvittavat tiedot haetaan palvelimelta AJAX- tai fetch-pyyntöinä. Sovelluslogiikasta siirretään siis ainakin osa JavaScriptinä selaimessa suoritettavaksi. Käyttäjän näkökulmasta SPA-sovellus on responsiivisempi ja toimii enemmän kuten tavallinen sovellus. Asiasta on kerrottu tarkemmin lähteessä [MP13].

2.3 Riskienhallinta

Tässä kappaleessa käydään läpi riskienhallintaan liittyviä peruskäsitteitä.

Uhka on mahdollinen ei-toivottu tilanne tai tapahtuma, joka voi aiheuttaa haittaa tai vahinkoa [Int18].

Riski on uhkan mahdollinen seuraus, potentiaalinen vahinko tai tappio uhkan toteutuessa. Riskin suuruus määräytyy uhkan seurausten ja todennäköisyyden perusteella. [Int18][Hom08]

Riskienhallinta on prosessi, joka käsittelee valittuun kohteeseen kohdistuvia riskejä. Prosessin tavoitteena on tunnistaa ja tiedostaa riskit, analysoida ja arvioida niiden vaikutuksia, tehdä tarvittavia korjaustoimenpiteitä riskien pienentämiseksi, sekä valvoa riskien toteutumista ja korjaustoimenpiteiden toimivuutta. Yleensä tavoitteena on pienentää riskin vaikutuksia, mutta riskin hyväksyminen ja vastuun siirtäminen ovat myös mahdollisia ratkaisuja. Vaikka riskien täydellinen ehkäiseminen ei yleensä ole mahdollista, voidaan usein tehdä toimenpiteitä riskin pienentämiseksi. [Int18][Hom08]

Haavoittuvuus on heikkous järjestelmässä tai sovelluksessa, jota voidaan hyväksikäyttää yhden tai useamman uhkan toteuttamiseen [Int18].

Hyökkäys on yritys tuhota, paljastaa, muokata, estää, varastaa, saada luvaton pääsy tai käyttää luvatta sovellusta tai järjestelmää. Hyökkäyksen voi toteuttaa yhden tai useamman haavoittuvuuden avulla. [Int18]

3 Web-sovellusten tietoturvariskit

Tässä luvussa käydään läpi web-sovellusten tietoturvariskejä OWASP Top 10 -listauksen avulla, sekä esitellään muita tyypillisiä web-sovellusten haavoittuvuuksia. Tämän tutkielman tarkoituksena ei ole käsitellä haavoittuvuuksien korjaustoimenpiteitä, vaan keskitytään haavoittuvuuksien havaitsemiseen.

3.1 OWASP Top 10

Open Web Application Security Project (OWASP) on maailmanlaajuinen voittoa tavoittelematon organisaatio, jonka tavoitteena on parantaa ohjelmistojen tietoturvaa. Sen tavoitteena on lisätä tietoisuutta tietoturvaongelmista ja niiden ratkaisuista, jotta henkilöt ja organisaatiot voivat tehdä valistuneempia päätöksiä tietoturvaratkaisuihin liittyen.

OWASP Top 10 on laajalti tunnettu listaus yleisimmistä web-sovellusten tietoturvariskeistä [OWA17]. Ne on valittu tutkimalla seitsemän web-sovellusten tietoturvaan erikoistuneen yrityksen keräämiä tietoja asiakkaittensa sovellusten haavoittuvuuksista. Lähteenä on käytetty tuhansia tutkittuja sovelluksia, joista löytyi yli 500 000 haavoittuvuutta. Top 10 on valittu ja järjestetty riskien yleisyyden, arvioitujen havaitsemisen ja hyväksikäyttämisen helppouden sekä mahdollisten seurausten vakavuuden perusteella.

OWASP Top 10:n tavoite on antaa tietoa sovelluskehittäjille, suunnittelijoille, arkkitehdeille, projektipäälliköille ja organisaatioille tärkeimmistä tietoturvariskien vaaroista.

OWASP Top 10 tutkii ja vertailee haavoittuvuuksia ja niihin liittyviä riskejä, mutta tässä työssä keskitytään tarkastelemaan OWASPin listaamia riskejä vain haavoittuvuuksina.

Listauksen viimeisimmän version julkaisusta on kulunut useampi vuosi, mutta uudempaa julkaisua ei ole vielä tehty. OWASP Top 10:n Github-versionhallinnassa [OWA20b] on viitteitä suunnitelmista julkaista päivitetty versio.

Taulukossa 3.1 on listattu OWASP Top 10:n tietoturvariskien hyväksikäytön helppous, yleisyys, tunnistettavuus, vaikutus sekä automaattisella skannauksella tunnistettavuus. Viimeinen sarake on lisätty tämän tutkimuksen tarkoitusta varten.

Riski	Hyväksikäytön helppous	Yleisyys	Tunnistettavuus	Vaikutus	Tunnistettavissa skannauksella
A1 - Injektiot	Helppo	Yleinen	Helppo	Vakava	Kyllä
A2 - Todentaminen	Helppo	Yleinen	Keskitaso	Vakava	Ei
A3 - Arkaluonteinen vuoto	Keskivaikea	Todella yleinen	Keskitaso	Vakava	Ei
A4 - XML-entiteetit	Keskivaikea	Yleinen	Helppo	Vakava	Kyllä
A5 - Rikkinäinen pääsynhallinta	Keskivaikea	Yleinen	Keskitaso	Vakava	Ei
A6 - Konfiguraatiovirheet	Helppo	Todella yleinen	Helppo	Kohtalainen	Osittain
A7 - XSS	Helppo	Todella yleinen	Helppo	Kohtalainen	Kyllä
A8 - Serialisointi	Vaikea	Yleinen	Keskitaso	Vakava	Osittain
A9 - Komponentit	Keskivaikea	Todella yleinen	Keskitaso	Kohtalainen	Osittain
A10 - Lokitus ja monitorointi	Keskivaikea	Todella yleinen	Vaikea	Kohtalainen	Ei

Taulukko 3.1: Yhteenveto OWASP Top 10 -listauksen riskeistä. Lainattu muokaten [OWA17].

Seuraavaksi käydään lyhyesti läpi yksitellen vuoden 2017 OWASP Top 10 -listauksen kymmenen kohtaa. Seuraavien kappaleiden lähteenä on [OWA17] ellei toisin mainita.

3.1.1 A1 - Injektiot

Injektiossa saadaan jonkin hyökkäyksen kohteena olevan sovelluksen käyttämä taustasovellus tai -kirjasto suorittamaan hyökkääjään antamia komentoja.

Kohdesovelluksen käyttäjä voi tehdä injektiohyökkäyksen yksinkertaisella tekstipohjaisella syötteellä. Syötettä tehdessä hyödynnetään tietämystä syötettä tulkitsevan taustasovelluksen syntaksista. Usein hyökkääjällä ei ole tietoa käytetyistä taustasovelluksista, jolloin saatetaan kokeilla yleisimmin käytettyjen ohjelmistojen syntakseja. Hyökkäyssyöte voidaan viedä sovellukseen lähes mitä kautta tahansa, esimerkiksi käyttöliittymän kautta, URL-parametrina, sovellukseen lähetettävänä tiedostona tai linkattuna hyökkääjän HTTP-palvelimelta.

Yleisimmin injektiohaavoittuvuuksia löytyy SQL, LDAP, Xpath, tai NoSQL -kyselystä, käyttöjärjestelmän komentorivin käytöstä, XML-jäsentimistä ja muista vastaavista käyttötapauksista, joissa sovellus välittää käyttäjän syötteen eteenpäin taustasovellukselle. Injektiosta voi aiheutua salaisten tietojen vuotamista, tietojen tuhoutumista, ohjelmiston käytön estymistä, tai hyökkääjä saattaa jopa saada ohjelmistoa ajavan koneen täysin haltuunsa.

Injektiohaavoittuvuudet ovat usein helposti havaittavissa tarkastelemalla sovelluksen koodia, mutta suuri osa niistä on myös mahdollista löytää testaamalla skanneriohjelmiston avulla [OWA17; Abe14].

3.1.2 A2 - Rikkinäinen todentaminen

Käyttäjien todentamiseen, eli sisäänkirjautumiseen, ja istunnonhallintaan liittyvät osat sovelluksessa on usein toteutettu virheellisesti. Tämän vuoksi hyökkääjä saattaa saada haltuunsa salasanoja, avaimia, *istuntotunnisteita* (session token) tai onnistua muuten esittäytymään järjestelmän toisena käyttäjänä.

Sovelluksiin räätälöidään usein käyttäjän todentamisen ja istunnonhallinnan toimintoja. Näiden komponenttien toteuttaminen virheettömästi on vaikeaa, joten niistä löytyy usein haavoittuvuuksia muun muassa uloskirjautumisesta, salasanojen hallinnasta, aikakatkaisuista, ”muista minut”-toiminnoista ja turvakysymyksistä. Hyökkäyksen onnistuessa hyökkääjä saa käyttöönsä yhden tai useampia muiden käyttäjien tunnuksista ja pystyy tekemään sovelluksessa toimintoja näiden käyttäjien oikeuksin. Tämän vuoksi hyökkäyksiä kohdistetaan yleensä etenkin ylläpitäjien käyttäjätileihin.

Todentamisen ja istunnonhallinnan haavoittuvuuksien löytäminen on toisinaan hankalaa, sillä jokainen räätälöity sovellus toimii eri tavalla. Siinä missä skannauksella löydettävissä olevat injektiohaavoittuvuudet voidaan yksilöidä ja listata, todentamisen ja istunnonhallinnan haavoittuvuudet riippuvat sovelluksen toteutuksesta, joten niitä ei pysty automaattisella skannauksella löytämään. Näiden haavoittuvuuksien löytämiseksi tarvitaan henkilö, joka voi hahmottaa, kuuluuko käyttäjän nähdä sovelluksen hänelle näyttämiä tietoja [Abe14].

3.1.3 A3 - Arkaluonteisen tiedon vuotaminen

Henkilötiedot ja muut arkaluonteiset tiedot ovat puutteellisesti suojattuja useassa sovelluksessa. Hyökkääjä voi esimerkiksi tehdä identiteettivarkauksia saamillaan henkilötiedoilla, tai käyttää hyödykseen luottokorttitietoja tai muita maksutietoja, joita sovelluksessa on tallennettu. Hyökkäyksen onnistuessa kaikki sovelluksen säilyttämät tiedot voivat päätyä hyökkääjän käsiin, joten kaikki arkaluonteinen tieto tulee säilyttää vain salatussa muodossa.

Kunnolla toteutetun salauksen murtaminen on todella aikaavievää, joten hyökkääjät pyrkivät ennemmin kaappaamaan tiedot niiden ollessa selväkielisessä muodossa. Käytännössä tämä tarkoittaa esimerkiksi salausavainten varastamista, välimieshyökkäystä siirrettäessä tietoja palvelimen ja käyttäjän selaimen välillä tai tietojen varastamista käyttäjän selaimesta. Yleisin arkaluonteisten tietojen vuotamiseen johtava virhe on tietojen salaamatta jättäminen. Salausta käytettäessä heikon salausalgoritmin valinta on yleinen virhe, etenkin valittaessa salasanojen tiivistealgoritmia.

Sen lisäksi, että tiedon arkaluonteisuutta on erittäin vaikeaa ohjelmallisesti päätellä, saattaa yhdessä sovelluksessa arkaluonteiseksi luokiteltava tieto olla toisessa sovelluksessa julkista tietoa. Vallitseva osa arkaluonteisten tietojen vuodoista ei siis ole havaittavissa skannamalla, mutta vuodon mahdollistavia haavoittuvuuksia saatetaan kylläkin skannamalla havaita [Abe14].

3.1.4 A4 - Ulkopuoliset XML-entiteetit

Ulkopuoliset XML-entiteetit (XML external entities) ovat XML-dokumentin osia, jotka viittaavat ulkopuoliseen tiedostoon tai URL-osoitteeseen, liittäen sen osaksi XML-dokumenttia. Mikäli XML-entiteettien käyttöä ei ole estetty, voi hyökkääjä esimerkiksi päästä lukemaan palvelimella olevia tiedostoja, päästä käsiksi palvelimen sisäverkossa oleviin palveluihin tai tehdä palvelunestohyökkäyksen.

Hyökkäysten estämiseksi tulee sovelluksen käyttämästä XML-kirjastosta konfiguroida XML-entiteetit pois käytöstä tai muuten varmistua siitä, ettei sovellukseen syötetyissä XML-tiedostoissa ole XML-entiteettejä. Näiden haavoittuvuuksien havaitseminen on mahdollista sekä konfiguraation staattisella analyysillä, että aktiivisella skannauksella.

3.1.5 A5 - Rikkinäinen pääsynhallinta

Kun sovellus näyttää käyttäjälle sovelluksen sisäisesti käyttämän viitteen, kuten tiedostonimen tai tietokantarivin tunnisteen, tapahtuu *suora viittaus olioon* (direct object reference). Mikäli sovellus ei tarkasta käyttäjän oikeuksia tällä tavalla viitattuihin kohteisiin, voi hyökkääjä muunnella näkemiään viitteitä ja tapahtuu *autentikoimaton suora viittaus olioon* (insecure direct object reference).

Sovellukset käsittelevät usein sisäisiä viitenumeroita ja tunnisteita myös käyttäjän selaimessa, jolloin ne ovat hyökkääjän nähtävissä ja muunneltavissa. Mikäli ne ovat tyypiltään juoksevia numerosarjota, tai muuten kontekstista helposti pääteltäviä arvoja, on hyökkääjän helppo arvata muita kelvollisia arvoja. Ilman kunnollista oikeustarkastusta hyökkääjä pääsee täten näkemään ja muokkaamaan tietoja, jotka eivät hänelle kuulu.

Kullekin sovelluksen käyttäjälle näytetään vain ne toiminnot ja sovelluksen osat, joihin hänellä on käyttöoikeus. Oikeustarkastukset tehdään usein juuri ennen kunkin toiminnon käyttöliittymäelementin näyttämistä. Näiden lisäksi on tärkeää tehdä oikeustarkastukset uudestaan palvelinpäässä, kun käyttäjä suorittaa toimintoja. Mikäli palvelinpäässä ei tehdä käyttöoikeuksien uudelleentarkastusta, hyökkääjä voi päästä käsiksi toimintoihin, mihin hänellä ei ole käyttöoikeuksia lähettämällä sovellukselle muunneltuja komentoja.

Toimintojen oikeustarkastuksen puuttuessa hyökkääjä voi päästä pahimmillaan ilman sisäänkirjautumista käsiksi rajattuihin toimintoihin muuntelemalla esimerkiksi sovelluksen käyttämiä URL-parametreja. Hyökkäyksiä kohdistetaan yleensä etenkin ylläpitäjän toimintoihin. Oikeustarkastukset puuttuvat useimmiten konfiguraatiovirheen takia, tai sovelluksen kehittäjän unohdettua toteuttaa käsin tehtävät tarkastukset.

Nämä haavoittuvuudet voi löytää sovelluskoodia tarkastelemalla tai manuaalisella testauksella. Rikkinäisen todentamisen tapaan näiden haavoittuvuuksien löytäminen vaatii testaajan, joka ymmärtää testattavan sovelluksen toimialan ja toiminnot. Automaattiset skannausohjelmistot eivät pysty tunnistamaan eroa käyttäjälle sallitun ja estetyn toiminnon välillä [OWA17; Abe14].

3.1.6 A6 - Väärin määritellyt tietoturva-asetukset

Hyvä tietoturva vaatii oikein määritellyt asetukset sovellukselle, sovelluksen kehykselle, web-palvelimelle, tietokantapalvelimelle, käyttöjärjestelmälle ja muille saman järjestelmän sovelluksille ja palveluille. Oletusasetukset eivät usein ole riittäviä, joten kaikki asetukset

tulee käydä huolella läpi. Käytettävät ohjelmistot tulee myös pitää ajan tasalla.

Huolimattomasti konfiguroidussa ympäristössä hyökkääjä saattaa päästä käsiksi järjestelmään oletustunnusten ja salasanojen avulla, tai saada muissa hyökkäyksissä avuksi olevaa tietoa järjestelmästä käyttämättömien ja suojaamattomien sivujen ja kansioiden avulla. Hyökkäyksen mahdollistava konfigurointivirhe voi piillä millä tahansa tasolla räätälöidystä sovelluksesta palvelimen käyttöjärjestelmään.

Automaattisella skannauksella voidaan tehokkaasti löytää suuri osa konfigurointivirheistä ja vanhentuneista ohjelmistoversioista, mutta testattavan sovelluksen tuntevan testaajan tulee tarkistaa, että skannauksen tuloksena olevat havainnot ovat todella tietoturvaaukkoja [OWA17; Abe14].

3.1.7 A7 - Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) -haavoittuvuuksissa sovellus lähettää käyttäjän syötteen takaisin selaimelle ilman *validointintia* (validation). Hyökkääjä voi täten ajaa ohjelmakoodia käyttäjän sovelluksessa, jolloin hän pystyy esimerkiksi kaappaamaan käyttäjän istunnon, muokkaamaan sovelluksen käyttäjälle näyttämää sisältöä tai ohjaamaan käyttäjän haluamalleen ulkopuoliselle sivustolle.

Syötteen validointi tarkoittaa esimerkiksi käyttäjän antaman tekstin tarkastamista, ennen kuin sitä käytetään paikoissa, jossa hyökkääjän muotoilema teksti voi aiheuttaa vahinkoa. Sovelluksien tulee aina olettaa, että syöte sisältää hyökkäyskoodia, joten kaikki syötteet tulee tarkastaa ja siistiä vaarattomiksi ennen käyttöä [MM14, s. 98].

XSS on yleisin web-sovellusten haavoittuvuustyyppi. XSS-haavoittuvuuksia on kolmea tyyppiä: *tallennettu* (stored), *heijastettu* (reflected) ja *DOM-pohjainen* (DOM based, document object model). Tallennetussa XSS-hyökkäyksessä ohjelmisto tallentaa hyökkäyskoodin tietokantaan ja näyttää sen sieltä myöhemmin hyökkäyksen kohteena oleville käyttäjille. Heijastetussa XSS-hyökkäyksessä hyökkäyskoodi ei tallennu palvelimelle, vaan se vaikuttaa vain koodin lähetyksen jälkeisellä HTML-sivulla. Hyväksikäyttääkseen heijastettua XSS-haavoittuvuutta täytyy hyökkääjän esimerkiksi saada uhri avaamaan hyökkäyskoodin sisältävä linkki sovellukseen, joten sen hyväksikäyttäminen on vaikeampaa, kuin tallennetun XSS-haavoittuvuuden. DOM-pohjaisessa XSS-hyökkäyksessä hyökkäyskoodi ei käy palvelimella lainkaan, vaan aktivoituu selaimessa sivun rakenteen muuttuessa

XSS-haavoittuvuuksien tunnistaminen on helppoa joko sovelluskoodia tarkastelemalla tai automaattisella skannauksella [OWA17; Abe14].

3.1.8 A8 - Sarjallistetun datan turvaton lukeminen

Monet ohjelmointikielet mahdollistavat olioiden sarjallistamisen tekstimuotoisiksi, mahdollistaen monipuolisemman siirtämisen ja käsittelyn. Mikäli sovellus saattaa käyttäjän nähtäville sarjallistetun olion, esimerkiksi HTTP-evästeessä tai HTML-lomakeparametreina, voi se mahdollistaa hyökkääjälle useita hyökkäyskeinoja sarjallistettua oliota muokkaamalla. Vakavimmillaan tämä voi johtaa hyökkääjän koodin suorittamiseen sovelluksessa.

Sarjallistetun datan turvatonta lukemista (insecure deserialization) onkin siis kaiken sellaisen sarjallistetun datan lukeminen, mitä hyökkääjä on saattanut päästä muokkaamaan.

Automaattisella skannauksella pystytään havaitsemaan käyttääkö sovellus käyttäjälle näkyviä sarjallistettuja olioita, mutta manuaalista työtä tarvitaan, jotta saadaan varmistuttua onko sarjallistamista käyttävä ohjelmisto haavoittuvainen.

3.1.9 A9 - Haavoittuvaisiksi tiedettyjen komponenttien käyttö

Kaikki web-sovellukset hyödyntävät muita sovelluksia ja komponentteja toiminnassaan, kuten sovelluskirjastoja, kehyksiä, sekä web- ja tietokantapalvelimia. Näitä komponentteja ajetaan lähes poikkeuksetta vähintään samoilla oikeuksilla kuin itse sovellusta, jolloin komponentin hallintaansa saava hyökkääjä voi päästä käsiksi kaikkeen sovelluksen aineistoon tai jopa saada koko palvelimen hallintaansa. Järjestelmä on aina vain yhtä turvallinen kuin sen heikoin komponentti. Yksittäisen komponentin tunnettu haavoittuvuus voi siis romuttaa koko järjestelmän suojaukset.

Hyökkääjä voi tunnistaa haavoittuvaisen komponentin automaattisella skannauksella tai käsin tehdyllä tarkastelulla. Yleisesti tunnettujen haavoittuvuuksien hyväksikäyttämiseen saattaa löytyä julkisia esimerkkikoodeja sekä apuohjelmistoja tai hyökkääjä voi kirjoittaa hyökkäyskoodinsa itse. Julkisesti näkyvien komponenttien hyväksikäyttäminen on helpompaa kuin sovelluksen sisäisesti käyttämien.

Lähes kaikki sovellukset käyttävät vanhentuneita komponentteja ja muiden komponenttien riippuvuuksina asennettavat komponentit lisäävät ennestään suurta ylläpitotaakkaa. Sovelluksen kehittäjät eivät välttämättä edes tiedä, millä minkäkin komponentin versiolla sovellusta tullaan käyttämään.

Haavoittuvaisesta komponentista ja itse haavoittuvuudesta riippuen hyökkäyksen seurauksina voi olla lähes mitä tahansa kosmeettisista virheistä koko palvelimen hallinnan menettämiseen.

Haavoittuvaisiksi tiedettyjen komponenttien havaitseminen onnistuu hyökkääjän näkökulmasta tähän käyttötarkoitukseen kohdistettujen automaattisten skannausohjelmistojen avulla. Sovelluksen ylläpitäjän kannattaa valvoa käytettyjen komponenttien versioita ja niistä löydettyjä julkaistuja haavoittuvuuksia [OWA17; Abe14]. Esimerkiksi Linux-pohjaisissa järjestelmissä Linux-jakelun pakettienhallinnasta löytyvien komponenttien käyttäminen on helppo tapa ulkoistaa tunnettujen haavoittuvuuksien päivittämisestä huolehtiminen pakettienhallinnan ja jakelun vastuulle.

3.1.10 A10 - Riittämätön lokitus ja monitorointi

Vaikka riittämätön lokitus ja monitorointi ei itsessään tarjoa hyökkääjälle uusia hyökkäyspintoja, se tarjoaa aikaa, mahdollisuutta kokeilla eri hyökkäyksiä, hyökätä naapurijärjestelmiin sekä ladata, muokata ja tuhota tietoja kenenkään huomaamatta.

Sovelluksen tulisi kirjoittaa lokille merkintöjä epäonnistuneista sisäänkirjautumisyrityksistä, käyttöoikeuspoikkeuksista ja validointivirheistä. Monitorointijärjestelmän tulisi hälyttää havaittuaan epäilyttävää toimintaa, jotta ylläpitäjät huomaisivat mahdollisen hyökkäyksen kohtuullisessa ajassa. Valitettavasti nämä tavoitteet eivät useinkaan täyty, sillä keskimääräinen aika hyökkäyksen havaitsemiseen on noin 200 päivää.

Lokituksen ja monitoroinnin toimivuutta ei automaattinen skannaus pysty havaitsemaan, mutta skannauksen tekemistä kokeiluista pitäisi jäädä lokeille ja monitorointiin selviä merkintöjä, joista voi päätellä hyökkäyksen olevan käynnissä.

3.2 Muita haavoittuvuuksia

OWASP Top 10:ssä mainittujen web-sovellusten haavoittuvuuksien lisäksi on olemassa muutamia muita yleisiä haavoittuvuuksia. Alla on mainittu muutamia, jotka eivät syystä tai toisesta mahtuneet OWASP Top 10:een, mutta ovat tämän tutkielman kannalta oleellisia. Osa näistä haavoittuvuuksista oli mukana OWASP Top 10:n aikaisemmassa, 2013 julkaistussa versiossa [OWA13].

3.2.1 Paikallisen tiedoston liittäminen

Paikallisen tiedoston liittämishaavoittuvuus (local file inclusion, LFI) sallii hyökkääjän liittää palvelimella olevan tiedoston ajettavaan sovellukseen. Tämä tapahtuu usein sovelluslo-

giikassa olevan dynaamisen tiedostoliitostoiminnon avulla puutteellisen käyttäjän syötteen validoinnin seurauksena. Käytännössä palvelin saa usein selaimelta syötteenä merkkijonon, joka vastaa palvelimella olevan, sovellukseen liitettävän tiedoston nimeä tai polkua. Puutteellisen validoinnin seurauksena hyökkääjä voi käskeä palvelinta liittämään haluamansa nimisen tiedoston tai polun esimerkiksi etsimällä yläkansioista "../"-merkinnöillä. Tämän vuoksi haavoittuvuutta kutsutaan myös *polun läpikulkemiseksi* (path traversal) [MM14, s. 142].

Hyökkäyksen seurauksena on usein liitetyn tiedoston sisällön näkyminen hyökkääjälle, mutta hyökkäys voi haavoittuvuuden vakavuudesta riippuen johtaa myös hyökkääjän koodin suorittamiseen palvelimella, *palvelun käytön estymiseen* (denial of service), arkaluonteisen tiedon vuotamiseen tai selaimessa ajettavan koodin muunteluun, mikä voi johtaa esimerkiksi XSS-hyökkäyksiin [MM14, s. 142]. Paikallisen tiedoston liittämishaavoittuvuus on tunnistettavissa automaattisella skannauksella.

3.2.2 Etätiedoston liittäminen

Etätiedoston liittämishaavoittuvuudessa (remote file inclusion, RFI) perusperiaate on sama kuin paikallisen tiedoston liittämishaavoittuvuudessa, mutta hyökkääjä pystyy antamaan palvelimen paikallisen polun sijaan toiseen palvelimeen viittaavan URL-osoitteen, josta hyökkäyksen kohteena olevan sovellus hakee tiedoston [MM14, s. 143].

Seurauksena hyökkäyksestä voi olla hyökkääjän koodin suorittaminen palvelimella tai selaimessa, palvelun käytön estyminen tai arkaluonteisen tiedon vuotaminen [MM14, s. 143]. Etätiedoston liittämishaavoittuvuus on tunnistettavissa automaattisella skannauksella.

3.2.3 Cross-Site Request Forgery (CSRF)

CSRF-hyökkäyksessä hyökkääjä huijaa sovellukseen sisäänkirjautuneen uhrin selaimen lähettämään sovellukselle hyökkääjän määrittelemän pyynnön. Mikäli sovellus on haavoittuvainen CSRF-hyökkäyksille, sovellus ei osaa tunnistaa, onko käyttäjältä tuleva pyyntö oikeasti käyttäjän vai hyökkääjän tekemä ja suorittaa pyynnön. Näin hyökkääjä pystyy suorittamaan toimintoja sovelluksessa uhrikäyttäjän oikeuksilla.

Hyökkääjä voi tehdä CSRF-hyökkäyksen esimerkiksi houkuttelemalla uhrin ylläpitämälleen sivustolle tai yhdistelemällä hyökkäyksessä kohdesovelluksen mahdollisia XSS-haavoittuvuuksia. CSRF-hyökkäyksen toteutusta varten hyökkääjä tarvitsee tuntemusta siitä

sovelluksen toiminnosta, jota hän yrittää saada uhrin suorittamaan. Tämä tieto on usein helposti saatavilla, mikäli hyökkääjällä on itsellään pääsy samaan sovelluksen toimintoon [OWA17].

CSRF-haavoittuvuuksien tunnistaminen on yleensä helppoa joko sovelluskoodia tarkastelemalla tai automaattisella skannauksella [OWA17; Abe14].

CSRF-hyökkäys oli mukana OWASP Top 10 -listauksen 2013-versiossa [OWA13].

3.2.4 Uudelleenohjausten puutteellinen validointi

Web-sovellukset hyödyntävät usein käyttöliittymässään uudellenohjauksia, joissa selainta käsketään siirtymään määrätylle sivulle. Ilman kunnollista validointia hyökkääjä saattaa pystyä kontrolloimaan, mille sivulle käyttäjä ohjataan tai hyökkääjä saattaa päästä käsiksi toimintoihin, joihin hänellä ei ole käyttöoikeuksia.

Hyökkääjä voi esimerkiksi huijata uhria klikkaamaan linkkiä, joka validoimattoman uudelleenohjauksen ansiosta vie käyttäjän eteenpäin hyökkääjän määrittämälle sivustolle. Koska linkki osoittaa uhrille tuttuun sovellukseen, saattaa hän todennäköisemmin napsauttaa sitä.

Sovelluksesta toiselle sivustolle ohjaavien uudellenohjausten haavoittuvuuden havaitseminen onnistuu automaattisella skannauksella, sillä osoitteessa näkyy usein uudelleenohjauksen kohdesivun osoite täydellisessä muodossaan. Sovelluksen sisäisten uudellenohjausten haavoittuvuuksien havaitseminen on vaikeampaa ja vaatii manuaalista testausta.

Uudelleenohjausten puutteellinen validointi oli mukana OWASP Top 10 -listauksen 2013-versiossa [OWA13].

4 Web-sovellusten tietoturvan testaus

Testaus on virheiden etsimistä sovelluksesta käyttämällä sitä. Kaikkien virheiden löytäminen testaamalla on kuitenkin käytännössä mahdotonta [MSB11, s. 6-9].

Kaksi vallitsevinta testausstrategiaa ovat *mustalaatikkotestaus* (black-box) ja *valkolaatikkotestaus* (white-box). Mustalaatikkotestauksessa sovellusta ajatellaan mustana laatikkona. Sovelluksen sisäisestä toiminnasta ja toteutuksesta ei tiedetä mitään. Testauksessa keskitytään siis löytämään sovelluksen toiminnasta tilanteita, jotka poikkeavat sovelluksen määrittelystä. Sovelluksen kaikenkattavaksi testaamiseksi täytyisi kokeilla, että sovellus hyväksyy kaikki mahdolliset oikeat syötteet ja että se hylkää kaikki virheelliset syötteet. Käytännössä tällöin pitäisi kokeilla kaikkien syötteiden kaikkia yhdistelmiä eli tehdä *tyhjentävä syötteiden testaus* (exhaustive input testing), joka yleensä tarkoittaa loputtoman suurta määrää vaihtoehtoja. Koska tämä on luonnollisesti mahdotonta, tulee mustalaatikkotestauksessa pyrkiä maksimoimaan löydettävien virheiden määrä käytettävissä olevilla testitapauksilla. Ensisijainen keino tähän on tutustua ohjelman toimintalogiikkaan testattavan komponentin osalta, jotta voidaan tehdä oletuksia mahdollisista virheiden riskialueista [MSB11, s. 9-11].

Mustalaatikkotestauksesta poiketen valkolaatikkotestauksessa keskitytään sovelluksen sisäisen rakenteen tarkasteluun. Valkolaatikkotestauksessa pyritään löytämään etenkin vaikeasti havaittavia virheitä. Käytännössä voidaan tutkia esimerkiksi sovelluksen koodin mahdollisia suorituspolkuja. *Yksikkötestaus* (unit testing) on yksi valkolaatikkotestauksen esiintymä, missä testataan ohjelman yksittäisiä moduuleita tai funktioita [DRP99, s. 239].

4.1 Web-sovellusten automaattinen testaus

Käytettäessä automaattista testausta kullekin testitapaukselle luodaan *testiohjelma* (test script), joka käyttää testattavaa sovellusta ja selvittää toimiiko testitapaus määrittelyn mukaisesti. Testiohjelmia ajetaan usein erillisellä *automaattisten testien työkalulla* (automated test tool). Testauksen automatisoinnin aiheuttamat kustannukset ovat pitkällä aikavälillä pienet verrattuna manuaaliseen testaukseen, kun testejä ajetaan toistuvasti. Etenkin sovelluksen kehityksen aikana testiajoja suoritetaan lukemattomia kertoja, jolloin automaattiset testit nousevat arvoonsa [DRP99, s. 4].

Sovelluksen kehittyessä myös sen testejä tulee muokata sovelluksen muutosten mukaan ja luoda uusia testejä testaamaan sovellukseen lisättyä toiminnallisuutta. Ajantasainen ja kattava testaaminen ovat olennainen sovelluksen oikeintoiminnan ja vakauden ilmaisin [DRP99, s. 4].

4.2 Web-sovellusten automaattinen tietoturvatestaus

Automaattiset tietoturvatestit voidaan jakaa kahteen osaan: sovelluskohtaisiin testeihin ja yleisiin testeihin. Sovelluskohtaiset testit luodaan nimensä mukaisesti testaamaan tiettyä sovellusta. Niiden etuna on mahdollisuus useampien ja vaikeammin havaittavien haavoittuvuuksien testaukseen. Sovelluskohtaisten testien tekeminen on kuitenkin työlästä ja aikaavievää.

Yleiset testit toimivat useissa testattavissa sovelluksissa ilman sovelluskohtaista konfigurointia. Niillä pyritään löytämään sovelluksesta tavallisia haavoittuvuuksia ilman, että sovelluksen rakenteesta ja toiminnalisuudesta tiedetään mitään [Che14]. Yleisten testien käyttötarkoitukset ovat rajatumpia, sillä monien haavoittuvuuksien testaamiseen tarvitaan sovelluskohtaista tietoa kontekstista ja toimintalogiikasta. Yleisten testien suuri etu on kuitenkin niiden uudelleenkäytettävyys. Tässä työssä käytetään yleisiä testejä ajavasta automaattisesta testausohjelmistosta nimitystä *skannausohjelmisto*.

Ilmaisia ja myös vapaan lähdekoodin skannausohjelmistoja on saatavilla kymmenittäin. Parhaimmillaan ne pystyvät löytämään hyvällä tarkkuudella muun muassa XSS-, CSRF-, uudelleenohjaus- ja injektiohaavoittuvuudet, sekä jotkin konfiguraatiovirheet ja osan sovelluksessa käytetyistä haavoittuvaiseksi tunnetuista komponenteista [Abe14; Las20a]. Nämä haavoittuvuudet muodostavat yhdessä noin puolet OWASP Top 10 -riskeistä, kuten taulukossa 3.1 on listattu. Yleisimmistä tietoturvariskiryhmistä on siis periaatteessa mahdollista tunnistaa puolet pelkällä automaattisella skannauksella ilman, että työtunteja joudutaan käyttämään sovelluskohtaisten konfiguraatioiden tai testien tekemiseen.

Koska tässä työssä pyritään tuottamaan projektien välillä helposti uudelleenkäytettävä ratkaisu tietoturvatestaukseen, sulkeutuu pois vaihtoehto sovelluskohtaisten testien käyttämisestä. Siispä työssä keskitytään yleisiä testejä käyttävien skannausohjelmistojen käyttöön.

4.3 Käytettävän testausohjelmiston valinta

Web-sovellusten haavoittuvuusskannereita on olemassa useita kymmeniä [Che14], joten kuhunkin käyttöön parhaiten soveltuvien työkalujen löytäminen näin suuren valikoiman joukosta voi olla haastavaa. Valikoiman rajaamiseksi voidaan tehdä ensiksi jako maksullisiin ja ilmaisiin sovelluksiin.

4.3.1 Sovellusten maksullisuus

Maksullisten ohjelmistojen etuja ovat valmistajan tarjoama tuki sovelluksen käyttöön ja mahdollisesti lisenssin mukana taatut päivitykset ohjelmistoon [McQ14]. Pelkästään siksi, että ohjelmistoja myydään maksullisina tuotteina, voidaan olettaa niissä olevan kattava tuki erilaisten haavoittuvuuksien havaitsemiseen. Maksullisten skannausohjelmistojen lisenssimaksut ovat muutamista sadoista dollareista jopa kymmeniin tuhansiin dollareihin vuodessa [Che14].

Ilmaisia ohjelmistoja käytettäessä on harvoin tarjolla maksullisia palveluita vastaavaa tukea ohjelmiston käyttöön. Ilmaisilla ohjelmistoilla on kuitenkin omat hyvät puolensa. Ensinnäkin ilmaisohjelmistoilla on helppo kokeilla, soveltuuko kyseinen ohjelmisto haluttuun käyttötarkoitukseen. Mikäli ohjelmisto ei vastaakaan odotuksia, voidaan tilalle ottaa huoletta toinen ohjelmisto, sillä lisenssiin ei olla sijoitettu rahaa. Toisekseen ilmaiset skannausohjelmistot ovat usein myös avointa lähdekoodia. Tämä voi olla joissain tapauksissa suuri etu, sillä avoin lähdekoodi mahdollistaa tarvittaessa ohjelman sisäisen toiminnan tutkimisen ja jopa muokkaamisen tarpeiden mukaan.

Avoimen lähdekoodin skannausohjelmistojen on havaittu olevan haavoittuvuuksien havaitsemistarkkuudeltaan samalla tasolla kaupallisten sovellusten kanssa tai jopa niitä edellä [McQ14]. Avoimen lähdekoodin skannerit eivät siis ole pelkästään kustannussyistä houkuttelevia, vaan kilpailevat samassa luokassa maksullisten ohjelmistojen kanssa.

Tässä luvussa esitettyjen haittojen ja hyötyjen perusteella tässä työssä keskitytään ilmaisiin, avoimen lähdekoodin skannausohjelmistoihin.

4.3.2 Ilmaisten ohjelmistojen suorituskykytestaus

Shay Chen vertailee web-sovellusten skannausohjelmistojen suorituskykytestissään kymmeniä kaupallisia ja ilmaisia ohjelmistoja [Che14]. Hän käyttää arviointiin WAVSEP-

testausalustaa (Web Application Vulnerability Scanner Evaluation Project), joka on suunniteltu arvioimaan web-sovellusten skannausohjelmistojen haavoittuvuuksien havaisemistarkkuutta. Testausalusta luo mittausasteikon, jonka avulla pyritään ymmärtämään, mitä yleisiä variaatioita haavoittuvuuksista kukin skanneri kykenee tunnistamaan. Testitapauksien joukossa on sekä yksinkertaisia, yleisiä että vaikeammin havaittavia haavoittuvuuksia [Che14].

Chen jaottelee käyttämänsä testitapaukset seuraaviin luokkiin:

- uudelleenohjauksen puutteellinen validointi
- piilotetut ja vanhentuneet tiedostot sekä varmuuskopiot
- paikallisen tiedoston liittäminen
- etätiedoston liittäminen
- heijastettu XSS
- SQL-injektiot.

Jokaisella luokalla on oikeiden testitapaustensa lisäksi joukko *väärien positiivisten* (false positive) -testejä, jotka saattavat esittää joitain potentiaalisten haavoittuvuuksien merkkejä olematta kuitenkaan oikeita haavoittuvuuksia. Niillä pyritään näkemään, kuinka tarkka testattavien skannereiden haavoittuvuuksien havaitsemisprosessi on [Che14].

Haavoittuvuustestien lisäksi Chen testaa ohjelmistojen *hyökkäyspinnan kattavuutta* (attack surface coverage) mittamalla ohjelmiston *etsijän* (crawler) tehokkuutta, testaamalla sen tukea erilaisille tekniikoille ja kykyä selviytyä moninaisista esteistä [Che14]. Etsijä on skannerin osa, joka käy läpi kohdesovelluksen läpi sivu sivulta, etsien aina viittauksia uusiin sivuihin, pyrkien näin kartoittamaan sovelluksen kaikki sivut ja toiminnot.

Skannausohjelmistot tarvitsevat tiedon kohdesovelluksen sivurakenteesta pystyäkseen löytämään kaikki eri sivuilla olevat haavoittuvuudet. Testaajan on joko annettava tämä tieto skannerille manuaalisesti, opetettava skanneria, tai skannerin etsijän on löydettävä sivut itsenäisesti. Skannerin opettaminen voi tapahtua esimerkiksi siten, että käyttäjä manuaalisesti selailee kohdesivustoa skannerin pitäessä kirjaa avatuista sivuista. Etsijällä on ensisijaisen tärkeä rooli käyttötapauksissa, joissa skanneri toimii itsenäisesti, sillä ilman kattavaa tietoa sivurakenteesta voi potentiaalisia haavoittuvuuksia jäädä testaamatta ja siten löytymättä [Che14].

Chen käyttää WIVET (Web Input Vector Extractor Teaser) -testaustyökalua skannereiden etsijöiden tehokkuuden testaamiseen. Skannaamalla WIVET-työkalun tarjoamaa websovellusta, työkalu pystyy laskemaan skannerin etsijän suhteellisen tehokkuuden [Che14].

Logo	Vulnerability Scanner	Benchmark Results						
	W3AF		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	19%	35.29%	37.88%	57.48%	16.67%	63.33%	22.83%
		False Positive		30.0%	0.0%	12.5%	16.67%	11.11%	0.0%
	arachni		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	96%	100.0%	90.91%	100.0%	100.0%	100.0%	100.0%
		False Positive		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
IRON WASP	IronWASP		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	✗	99.26%	100.0%	53.06%	77.78%	73.33%	✗
		False Positive		0.0%	0.0%	0.0%	0.0%	11.11%	✗
	ZAP		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	73%	100.0%	100.0%	75.0%	100.0%	16.67%	38.04%
		False Positive		30.0%	0.0%	0.0%	16.67%	0.0%	33.33%
	Syhunt Mini (Sandcat Mini)		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	✗	100.0%	100.0%	✗	44.44%	✗	18.48%
		False Positive		50.0%	0.0%	✗	0.0%	✗	100.0%
	SkipFish		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	48%	76.47%	93.94%	82.35%	31.48%	36.67%	✗
		False Positive		0.0%	0.0%	25.0%	16.67%	0.0%	✗
	Wapiti		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	44%	100.0%	66.67%	51.47%	57.41%	✗	4.35%
		False Positive		20.0%	42.86%	12.5%	0.0%	✗	100.0%
	Sandcat Free Edition		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	✗	58.82%	98.48%	✗	✗	✗	✗
		False Positive		20.0%	85.71%	✗	✗	✗	✗
	Vega		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	50%	100.0%	100.0%	94.12%	100.0%	✗	✗
		False Positive		20.0%	0.0%	62.5%	0.0%	✗	✗
	Grendel Scan		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	14%	42.65%	12.12%	✗	✗	✗	✗
		False Positive		50.0%	0.0%	✗	✗	✗	✗
	WATOBO		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup
		Accuracy	1%	83.09%	75.76%	41.18%	✗	✗	11.96%
		False Positive		60.0%	100.0%	0.0%	✗	✗	100.0%

Taulukko 4.1: Web-sovellusten ilmaisten skannausohjelmistojen suorituskykytestin tuloksia. Lainattu muokaten [Che15].

Vuosi Chenin artikkelin julkaisun jälkeen päivitetyissä tilastoissa nähdään Arachnin pärjänneen selvästi parhaiten avoimen lähdekoodin skannereista [Che15]. Ote parhaiten vertailussa pärjänneiden ilmaisten skannausohjelmistojen tuloksista on nähtävissä taulukossa 4.1. Taulukosta voidaan nähdä Arachnin päässeen hyvin lähelle täydellistä tulosta WIVET ja WAVSEP-testeissä. Arachnin neljän prosentin vaje WIVET-testissä johtuu SWF-tiedostojen, eli Flash-tuen puutteesta ja noin yhdeksän prosentin puutos heijastettujen XSS-haavoittuvuuksien tuloksessa johtuu VBScript-tuen puutteesta. VBScript on vanhentunut tekniikka, jota tukevat vain vanhemmat versiot Internet Explorer-selaimesta [Las20b]. Käytännössä Arachni sai siis testeissä täydet pisteet relevanteilta osin.

Muita hyvin pärjänneitä vaihtoehtoja ovat IronWASP, ZAP ja Subgraph Vega. Käyttökelpoisia vaihtoehtoja karsii yrityksen ajoympäristön asettamat rajoitteet, sillä skanneria on tarkoitus ajaa komentoriviltä Linux-pohjaisella palvelimella.

IronWASP on Windows-sovellus, joten sitä ei pystytä ajamaan kohdeympäristössä [Kup20]. Subgraph Vega taas on graafinen Java-sovellus, jota ei pysty käyttämään komentoriviltä [Sub20]. Myös ZAP on Java-sovellus, joka on tarkoitettu pääasiassa graafisen käyttöliittymän kautta käytettäväksi, mutta sisältää myös mahdollisuuden skannauksen käynnistämiseen komentoriviltä [Ben20a].

Chenin vertailun tilastoja [Che15] ei ole päivitetty vuoden 2016 jälkeen, mutta muiden lähteiden perusteella tilanne ei ole sen jälkeen juuri muuttunut, eikä uusia varteenotettavia skannausohjelmistoja ole ilmaantunut [Idr+17] [KQS18].

Arachni on ensisijaisesti komentoriviltä käytettävä sovellus, joten se sopii lähtökohdiltaan hyvin tarvitsemaamme käyttötapaukseen [Las20a]. Arachni sai myös testin parhaimman tuloksen WIVET-testissä, mikä mittaa käyttötapauksessamme tärkeää etsijän tehokkuutta. Arachni suoriutui muissakin testeissä kautta linjan loistavasti ja selviytyi ilman yhtäkään false positive -osumaa [Che15]. Arachni on myös aikaisemmin menestynyt hyvin vastaavissa testeissä [SE14; McQ14].

Myös ZAP tarjoaa komentoriviltä ajettavan automaattisen skannauksen [Ben20a]. Toisin kuin Arachnia, ZAPia on kehitetty aktiivisesti myös viime vuosina. Tässä luvussa esitetyillä perusteilla valitsemme Arachnin ja ZAPin tässä työssä käytettäviksi skannausohjelmistoiksi.

SPA-sovellusten testaus

Chenin vertailussaan käyttämät WIVET- ja WAVSEP-testit sisältävät yksinkertaisia Javascript-testitapauksia, mutta ne eivät ole tarpeeksi kattavia SPA-sovelluksia vastaavan käytöksen testaamiseen.

SPA-sovellukset ja sovellusten vahva riippuvaisuus Javascriptistä ovat verrattain uusia trendejä, joihin skannaussovellukset eivät ole vielä kovin hyvin sopeutuneet. Edmund Foster vertailee kolmen skannausohjelmiston toimivuutta Javasciptiä käyttävään Hackazon-testisovellukseen [Fos18]. Ohjelmistot löysivät testisovelluksessa olevia haavoittuvuuksia vaihtelevasti, mutta vaikka kaikkien kolmen tulokset laskettaisiin yhteen, jäi silti 75% haavoittuvuuksista havaitsematta. Fosterin päätelmien perusteella automaattiset skannausohjelmistot eivät ole vielä valmiita SPA-sovellusten kunnolliseen haavoittuvuustarkistukseen. Jos skannausohjelmistoja kuitenkin käytetään SPA-sovellusten haavoittuvuustarkistukseen, hän suosittaa usean ohjelmiston samanaikaista käyttöä kattavampien tulosten saamiseksi.

OWASP Juice Shop [OWA20a] on moderni, Angular-kirjastoa [Ang20] käyttäen toteutettu SPA-sovellus, joka sisältää mahdollisimman paljon erilaisia haavoittuvuuksia. Se on tarkoitettu haavoittuvuusskannerien testauskohteeksi sekä manuaalisen penetraatiotestauksen harjoittelukohteeksi. Juice Shop sisältää yksittäistapauksia kaikista OWASP Top 10:ssä listatuista haavoittuvuuksista, sekä lisäksi muita oikeassa web-sovelluksissa vastaantulevia haavoittuvuuksia.

Arachnin ja ZAPin automaattiskannauksia testattiin Juice Shopia vasten. Jotta testausympäristö pysyisi mahdollisimman yksinkertaisena, testit ajettiin Dockerissa [Doc20]. Sekä Juice Shopista että molemmista skannausohjelmistoista on tarjolla valmiit Docker-levykuvat. Juice Shop -sovellus ja Arachnin sekä ZAPin skannaukset käynnistettiin listauksessa 4.1 olevilla komennoilla.

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
docker run --rm -v $(pwd):/zap/wrk/:rw -t owasp/zap2docker-weekly zap-full
   -scan.py -t http://host.docker.internal:3000 -r testreport.html
docker run --rm -v $(pwd):/root/:rw -t arachni/arachni /usr/local/arachni
   /bin/arachni http://host.docker.internal:3000 -report-save-path=/root/
   testreport.afr
docker run --rm -v $(pwd):/root/:rw -t arachni/arachni /usr/local/arachni
   /bin/arachni_reporter /root/testreport.afr --reporter=html:outfile=/
   root/testreport.html.zip
```

Listaus 4.1: Komennot Juice Shopin, sekä Arachnin ja ZAPin skannausten käynnistykseen.

Kuten taulukosta 4.2 nähdään, kumpikin skanneri löysi Juice Shopin noin sadasta haavoittuvuudesta vain hyvin pienen osan. Kriittisin Arachnin löytämä havainto oli SQL-injektio. ZAPin löytämistä havainnoista kriittisin oli CSRF. Keskitason havainnoista oleellisin Arachnilla oli validoimaton uudelleenohjaus ja ZAPilla varmuuskopion paljastuminen. Molemmat löysivät lisäksi muutamia ympäristön konfigurointivirheitä, jotka eivät varsinaisesti koske itse testattavaa sovellusta, eivätkä siten ole tämän työn kannalta kiinnostavia.

Sovellus	**Korkea**	**Keskitaso**	**Matala**	**Info**
Arachni	1	3	3	1
ZAP	1	5	2	3

Taulukko 4.2: Arachnin ja ZAPin löytämät havainnot eri riskitasoilla Juice Shopista.

Löytyneiden havaintojen pienen määrän ja tyyppien perusteella näyttää siltä, etteivät skannerit kykene juurikaan löytämään SPA-sovelluksessa olevia haavoittuvuuksia. Koska nämä skannerit eivät sovellu SPA-sovellusten testaamiseen, mutta Chenin tulosten perusteella löytävät hyvin haavoittuvuuksia perinteisistä web-sovelluksista, keskitytään tässä työssä perinteisten web-sovellusten testaamiseen.

4.4 Valittujen testausohjelmistojen esittely

Tässä luvussa esitellään lyhyesti käytettäväksi valitut ohjelmistot.

4.4.1 Arachni

Ensimmäiseksi tässä työssä käytettäväksi skannausohjelmistoksi valittu Arachni on monia ominaisuuksia sisältävä, modulaarinen Ruby-kielellä toteutettu web-sovellusten penetraatiotestauskehys. Se on suunnattu erityisesti tietoturvatestaajien ja sovellusten ylläpitäjien käyttöön helpottamaan web-sovellusten tietoturvan tason arviointia [Las20a].

Arachni on avointa lähdekoodia, mikä mahdollistaa muun muassa skannerin sisäisen toiminnan tarkastelun sekä sen toimintojen muokkaamisen. Skannerin sisäisen toiminnan tarkastelu ja tutkiminen voi lisätä luottoa työkalun antamien tulosten oikeellisuuteen, jos skannauksen tuloksia käytetään esimerkiksi olennaisena metriikkana skannattavien sovellusten tietoturvan tason määrittämisessä. Skannerin muokkaaminen puolestaan mahdollistaa muun muassa skannerin ajamisen ja kontrolloimisen erikoisemmissakin ympäristöissä, skannerin luomien raporttien räätälöimisen ja mukautettujen testien tekemisen [Las20a].

Arachnin käyttäminen onnistuu neljällä eri tavalla: komentoriviltä, erillisestä web-käyttöliittymästä, Ruby-kirjastona tai RPC-protokollalla (remote procedure call). Kaikillä käyttötavoilla käyttöönotto on yksinkertaista, sillä Arachnilla ei ole ulkopuolisia riippuvuuksia tietokantoihin tai kirjastoihin, eikä erillistä konfigurointia tarvita [Las20a].

Arachnissa on runsaasti testejä erilaisten haavoittuvuuksien havaitsemiseen. Vertailtaessa havaittujen haavoittuvuusryhmien määrää Chenin suorituskykytestiin valittujen ohjelmistojen kesken, Arachni oli toiseksi paras ilmaisten ohjelmistojen joukosta ja kuudes kaikista ohjelmistoista [Che15]. Arachnin tunnistamia haavoittuvuuksia ovat muun muassa erilaiset injektiot, CSRF, XSS, uudelleenohjausten validoinnin puutteet, etätiedostojen ja paikallisten tiedostojen liittäminen sekä jotkin konfigurointivirheet [Las20a].

Perinteisten web-sovellusten lisäksi Arachni hallitsee myös modernien JavaScript-painotteisten sivujen skannauksen, sillä se kykenee seuraamaan *tiedon ja suorituksen kulkua* (data and execution flows) sekä DOM-, että JavaScript-ympäristöissä. Arachni tunnistaa myös JQuery ja AngularJS-kirjastojen käytön ja optimoi suoritettavia testejä näitä kirjastoja varten. Haavoittuvuuden löytyessä Arachni tarjoaa *suorituspinon* (stacktrace), suoritettavan funktion nimen, sijainnin, lähdekoodin ja argumentit ohjelman tilasta haavoittuvuuden havaitsemisen hetkellä. Tämä helpottaa ja nopeuttaa ongelman selvitystä

etenkin vahvasti JavaScriptin varassa toimivissa web-sovelluksissa [Las20a].

Arachni analysoi skannattavaa ohjelmaa skannauksen aikana ja päättelee siinä käytössä olevia tekniikoita. Tämän tiedon perusteella se rajaa kyseiselle sovellukselle turhia testejä pois käytöstä, mikä vähentää skannauksessa tarvittavien HTTP-pyyntöjen määrää ja nopeuttaa skannauksen valmistumista. Lisäksi Arachni tutkii kohdesovelluksen palauttamia sivuja koko skannauksen ajan ja tunnistaa niistä muun muassa räätälöidyt 404-virhesivut, seuraa kohdepalvelimen suorituskykyä ja tunnistaa uusia *hyökkäyskohteita* (input vectors) [Las20a].

Skannausohjelmistot joutuvat tekemään valtavan määrän HTTP-pyyntöjä kohdesovellukseen testauksen aikana. Pienetkin viiveet yksittäisessä pyynnössä kumuloituvat äkkiä ja näkyvät selvästi skannauksen kokonaisajassa. Arachni pyrkii nopeuttamaan skannausaikoja tekemällä HTTP-kutsut asynkronisesti ja suorittamalla JavaScriptin ja muut selaimen toiminnot rinnakkain useassa prosessissa [Las20a].

Skannauksen valmistuttua Arachni luo sen tuloksista raportin, joka sisältää runsaasti tietoa havaittujen haavoittuvuuksien yksityiskohdista ja tarvittavat tiedot havaintojen toistamiseen. Näitä tietoja ovat muun muassa haavoittuvuuden sisältäneen sivun HTTP-pyyntö ja vastaus, sivun latauksen jälkeen tapahtuneet muunnokset sivun rakenteessa, tieto sivun latauksen aikana tapahtuneista HTTP-pyynnöistä ja tarkat tiedot ajossa olevista JavaScript-koodeista haavoittuvuuden havaitsemisen hetkellä. Raportin voi muuntaa useaan muotoon, kuten XML-, JSON-, HTML-, tai tekstimuotoiseksi. HTML-muotoisessa raportissa löydettyjen haavoittuvuuksien tilastot on esitetty myös havainnollistavina graafisina esityksinä [Las20a].

4.4.2 ZAP

Toiseksi tässä työssä käytettäväksi skannausohjelmistoksi valittu OWASP ZAP (Zed Attack Proxy) on Javalla toteutettu penetraatiotestaustyökalu. Se sisältää toiminnot itsenäiseen automaatiotestaukseen, mutta voi toimia myös manuaalisen testauksen apuvälineenä. [Ben20b]

ZAP toimii välityspalvelimena testaajan selaimelle, jolloin se pääsee näkemään liikenteen kohdesovellukseen ja muokkaamaan sitä tarpeen mukaan. Itsenäisessä automaatioskannauksessa ZAPilla on käytettävissään kaksi eri metodia kohdesovelluksen läpikäyntiin. Perinteinen sivuston läpikäynti etsii linkkejä sovelluksen antamista HTML-vastauksista. Sen toiminta on nopeaa, mutta se ei ota huomioon JavaScript-toiminnallisuutta. JavaScriptiä

hyödyntäville sovelluksille kannattaa käyttää AJAXia hyödyntävää sivuston läpikäyntiä, joka avaa läpikäytävät sivut selaimessa. Docker-kontissa ajettaessa hyödynnetään Firefox-selainta headless-tilassa. Tässä tilassa sivuston läpikäynti on hitaampaa, mutta tulokset ovat usein parempia. [Ben20b]

Sivuston läpikäynnin aikana ZAP yrittää löytää haavoittuvuuksia passiivisilla hyökkäyksillä tutkimalla sovelluksen antamia vastausviestejä. Läpikäynnin jälkeen alkaa aktiivinen hyökkäysosuus, jossa ZAP yrittää tunnettujen hyökkäysten avulla löytää haavoittuvuuksia kohdesovelluksesta. Hyökkäysten suorituksen jälkeen ZAP luo testeistä vastaavan HTML-raportin kuin Arachni. [Ben20b]

5 Tietoturvatestaus osana sovelluksen kehitystä

Tässä osiossa kuvaillaan, millaisessa ympäristössä tutkimus tehdään ja miten automaattinen tietoturvatestaus otetaan käyttöön.

5.1 Tilanne ennen testausta

Tutkimusympäristönä käytetään Oikeat Oliot Oy:n [Oli20] GitLabia [Git20e] keskitettynä versionhallintana, sekä sen CI/CD (Continuous Integration / Continuous Deployment) -ominaisuuksia sovellusten kääntämiseen, paketointiin, testaamiseen ja käyttöönottoon. GitLab-instanssi on yrityksen ylläpitämä ja se sijaitsee sen palvelinsalissa.

Jokaisen versionhallintaan viedyn sovellusmuutoksen jälkeen GitLab CI/CD kääntää automaattisesti sovelluksesta uuden version ja ajaa sille mahdolliset automaatiotestit. Kustakin sovelluksesta on aina käynnissä yleinen testi-instanssi manuaalista testausta varten. GitLab päivittää käännöksen ja testien päätteeksi tähän yleiseen testi-instanssiin uuden sovellusversion.

5.2 Toteutus

Automaattinen tietoturvaskannaus on mahdollista lisätä osaksi sovellusten *käännösputkea* (pipeline), mutta koska tietoturvaskannauksessa saattaa vierähtää useita tunteja, se ei ole käytännöllinen ratkaisu. Suoritetaan sen sijaan automaattiset tietoturvaskannaukset viikoittain, ajastettuna alkamaan lauantain ja sunnuntain välisenä yönä. Viikonloppuyönä tapahtuu vähiten sovelluskehitystä, joten silloin pitkäkestoiset tietoturvaskannaukset häiritsevät mahdollisimman vähän muuta työskentelyä.

```
vulntest-arachni:
  only:
    - schedules
    - web
  image: tiangolo/docker-with-compose:latest
  services:
    - docker:dind
  before_script:
    - cd src/vulntest
    - docker-compose -p vulntest_sovellus up -d
    - docker run --rm --network vulntest_sovellus willwill/wait-for-it -t
       1800 web:8080
  script:
    - docker run --rm -v $(pwd):/root/:rw -t --network vulntest_sovellus
      arachni/arachni /usr/local/arachni/bin/arachni http://web:8080/
      --report-save-path=/root/testreport.afr
      --plugin=autologin:url=http://web:8080/login,parameters="j_username
        =yllapitaja&j_password=salasana",check="Kirjaudu␣ulos" --scope-
        exclude-pattern=logout
    - docker run --rm -v $(pwd):/root/:rw -t arachni/arachni /usr/local/
      arachni/bin/arachni_reporter /root/testreport.afr --reporter=html:
      outfile=/root/testreport.html.zip
  after_script:
    - cd src/vulntest
    - docker-compose -p vulntest_sovellus down
  artifacts:
    paths:
      - src/vulntest/testreport.afr
      - src/vulntest/testreport.html.zip
```

Listaus 5.1: .gitlab-ci.yml -esimerkki Arachnin skannaustehtävän määrittelystä.

```
vulntest-zap:
  only:
    - schedules
    - web
  image: tiangolo/docker-with-compose:latest
  services:
    - docker:dind
  before_script:
    - cd src/vulntest
    - docker-compose -p vulntest_sovellus up -d
    - docker run --rm --network vulntest_sovellus willwill/wait-for-it -t
       1800 web:8080
  script:
    - docker run --rm -v $(pwd):/output:rw -t --network vulntest_sovellus
       registry.gitlab.com/gitlab-org/security-products/dast:latest
      /analyze -t http://web:8080 -j --full-scan 1 -r testreport.html
      --auth-url http://web:8080/login
      --auth-username 'yllapitaja'
      --auth-password 'salasana'
      --auth-username-field 'j_username'
      --auth-password-field 'j_password'
      --auth-submit-field 'login-submit'
      --auth-exclude-urls 'http://web:8080/html?command=logout'
      --exclude-rules 41,42,43,10027,10045,20017
  after_script:
    - cd src/vulntest
    - docker-compose -p vulntest_sovellus down
  artifacts:
    paths:
      - src/vulntest/testreport.html
```

Listaus 5.2: .gitlab-ci.yml -esimerkki ZAPin skannaustehtävän määrittelystä.

Tietoturvaskannaukset toteutetaan GitLab CI/CD:n tehtävinä. Projektikohtaisiin .gitlab-ci.yml -tiedostoihin lisätään erilliset tehtävät Arachnin ja ZAPin skannausajoista. Lis-

tauksissa 5.1 ja 5.2 on esitetty esimerkit tehtävämäärittelyistä. Molemmat skannausajot toimivat samalla idealla: sovelluksesta käynnistetään uusi paikallinen instanssi tietoturvaskannausta varten, suoritetaan skannaus ja lopuksi tuhotaan alussa käynnistetty sovellusinstanssi tietokantoineen.

Tietoturvaskannaukset olisi mahdollista ajaa aina käynnissä olevaa sovelluksen yleistä testi-instanssia käyttäen, mutta erillisen ympäristön luomisella saadaan useita hyötyjä. Erillistä testausinstanssia käyttämällä varmistutaan siitä, että sovelluksen tietokannassa on tunnettu sisältö, ja että kannan sisältö ei muutu ajokertojen välissä. Lisäksi skannausohjelmistot luovat testiensä aikana tietokantaan uusia rivejä, joita ei haluta yleiseen testiympäristöön häiritsemään manuaalista testausta.

GitLab CI/CD -tehtävissä käytetään Docker-in-Docker -palvelua (`docker:dind`), jotta tehtävästä käsin voidaan ajaa Docker-komentoja. Komento `docker-compose up` käynnistää paikallisen instanssin testattavasta sovelluksesta ja sen tietokannasta, `wait-for -it` -kontin avulla odotetaan tietokannan ja sovelluksen käynnistymistä, `docker run` -komennoilla käynnistetään tietoturvaskannaus ja sen päätyttyä `docker-compose down` -komennolla suljetaan ja poistetaan sovellus tietokantoineen. Skannerin luoma raportti on ladattavissa tehtävän suorittamisen jälkeen `artifacts`-määrittelyn avulla.

Molemmille skannausohjelmistoille annetaan `docker run` -komennossa konfiguraatiovipuja tehokkaamman skannauksen ja paremman tuloksen saavuttamiseksi. ZAPista käytetään GitLabin DAST-nimellä paketoimaa varianttia, joka tarjoaa helpomman tavan konfiguroida kirjautumistietoja. Näillä `--auth` -alkuisilla vivuilla kerrotaan, missä sisäänkirjautumissivu sijaitsee, millä tunnuksilla skanneri voi kirjautua sovellukseen ja mitä uloskirjautumislinkkejä skannerin tulee välttää. Muita käytettyjä konfiguraatiovipuja ovat AJAX-etsintä (`-j`) ja aktiivinen skannaus (`--full-scan`). Arachnille vastaavat kirjautumistiedot annetaan autologin-liitännäisellä. Arachni käyttää oletusarvoisesti kaikkia tuntemiaan skannaustekniikoita, joten muita konfiguraatiovipuja ei tarvita. Arachni kirjoittaa skannauksen tulokset AFR-tiedostoon, joka saadaan muunnettua HTML-muotoiseksi `arachni_reporter`-komennolla.

Tehtävät ajastetaan käynnistymään GitLabin *käännösputken ajastustoiminnon* (pipeline schedule) avulla [Git20b]. Käännösputken ajastuksessa on valittavissa valmis ”joka viikko”-valinta, mutta sen tarjoama ajankohta ei osu viikonlopulle. Käytetään siis räätälöityä cron-ajastusta, jolloin arvolla `0 2 * * 6` saadaan ajo käynnistettyä joka lauantai kello 2 yöllä.

5.2.1 GitLab CI:n konfigurointi

Skannausajon konfiguraation lisäksi GitLab-instanssin asetuksissa täytyy huomioida muutamia asioita.

Oletuksena yksittäisellä CI-ajolla on yhden tunnin aikakatkaisu. Koska skannausajot voivat kestää huomattavasti pidempään, täytyy tätä rajaa kasvattaa. Aikakatkaisurajan muuttaminen onnistuu projektikohtaisesti GitLabin web-käyttöliittymältä projektin asetuksista. Tämän työn testitapauksille uudeksi arvoksi asetettiin 48 tuntia.

GitLab CI:n ajoja suoritetaan GitLabin *ajajalla* (runner) [Git20c]. Ajaja on prosessi, joka huolehtii CI-ajojen suorittamisesta ja palauttaa ajon tulokset GitLab CI:lle, jonka web-käyttöliittymältä niitä voidaan seurata. Ajaja voi olla asennettuna eri koneelle kuin GitLab-instanssi ja se voidaan asentaa joko perinteisenä sovelluksena tai Docker-konttina.

Ajajalla täytyy olla vähintään yksi *suorittaja* (executor) [Git20d]. Suorittajat ovat erilaisia ajoympäristöjä, joissa ajot suoritetaan. Mahdollisia suorittajaympäristöjä ovat mm. Shell, SSH, Docker ja Kubernetes.

Tässä työssä käytetään Docker-suorittajaa, koska se on verrattain pienitöinen pystyttää, Dockerissa ajettavia skannausajoja varten Docker-ympäristö tarvitaan joka tapauksessa ja se mahdollistaa saman testiajon ajamisen useaan kertaan samanaikaisesti.

Skannausajoissa käytetään Dockeria, joten Docker-suorittajaan täytyy konfiguroida Docker-in-Docker-tuki. Oikeiden Olioiden testiympäristössä Docker-in-Docker oli alun perin mahdollistettu jakamalla isäntäkoneen *Docker-pistoke* (Docker-socket) *levynä* (volume) suorittajakontille. Tällä tavalla konfiguroituna CI-ajossa ei kuitenkaan ole mahdollista hyödyntää levyliitoksia. Niitä tarvitaan testiajoissamme sekä sovelluskonfiguraatioiden asettamiseen että ajon tulostiedostojen talteen ottamiseen, joten tämä konfiguraatio ei sovellu skannausajojen suoritukseen. [Git20a]

```
[[runners]]
  name = "docker-runner"
  url = "https://git.oikeatoliot.fi/"
  token = "..."
  executor = "docker"
  environment = ["DOCKER_TLS_CERTDIR=/certs/${CI_JOB_ID}", "DOCKER_DRIVER
     =overlay2"]
  [runners.docker]
    image = "debian:buster"
    privileged = true
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache", "/tmp/gitlab/docker/certs:/certs", "/dev/shm:/dev
       /shm"]
    pull_policy = "always"
    shm_size = 0
```

Listaus 5.3: GitLab Runnerin konfiguraatioesimerkki.

Vaihdetaan isäntäkoneen Docker-pistokkeen sijaan käyttöön Docker-in-Docker-palvelu. Muutoksen jälkeinen GitLabin ajajan konfiguraatio on listauksessa 5.3. Oleellisimmat asetukset tässä ovat:

- `executor = "docker"`, käytetään Docker-suorittajaa.
- `privileged = true`, ajetaan Dockeria luottamuksellisessa tilassa. Tämä mahdollistaa laajemmat pääsyoikeudet isäntäjärjestelmään, mikä voi olla tietoturvariski. Luottamuksellinen tila kuitenkin tarvitaan Docker-in-Dockerin toimimiseksi.
- `volumes = ["/cache", "/tmp/gitlab/docker/certs:/certs", "/dev/shm:/dev /shm"]`, annetaan konteille pääsy `/certs`-levyyn, jonka avulla kontin sisältä saadaan otettua TLS-salattu yhteys Docker-palveluprosessiin.
- `environment = ["DOCKER_TLS_CERTDIR=/certs/${CI_JOB_ID}", "DOCKER_DRIVER =overlay2"]`, kerrotaan kontissa ajettavalle Docker-asiakasohjelmalle mistä sertifikaatti TLS-salattua yhteyttä varten löytyy.

5.3 Toteutuksen tehokkuus

Oletusasetuksilla sekä Arachnin että ZAPin skannausajot kestävät epäkäytännöllisen kauan. Ensimmäisessä testattavassa sovelluksessa oletusasetuksilla ZAP-ajo kesti noin kuusi tuntia. Oletusasetuksilla Arachni suoritti skannausta yli kymmenen tuntia, minkä jälkeen kaikki pyynnöt päätyivät aikakatkaisuvirheeseen. Testejä ajettaessa Arachnista oli käytössä versio 1.5.1 ja ZAPista versio D-2020-06-30 (DAST v1.28.0).

Molemmille sovelluksille on mahdollista antaa ajon käynnistyksen yhteydessä asetusparametreja, joilla säädellään ajon toimintaa ja suoritettavia testejä. Pyritään siis löytämään molemmille sovelluksille sellaiset parametristot, joilla ajoihin kuluva aika saadaan minimoitua, ilman vaikutusta ajosta saataviin olennaisiin tuloksiin.

Arachnille hyödyllisiä parametreja olivat:

- `--platforms=linux,sql,mysql,tomcat,java`, joka kertoo kohdesovelluksessa käytetyn ohjelmointikielen, sekä palvelimen ja varusohjelmistojen tyypit, jotta muille tuotteille kohdennetut hyökkäykset voidaan jättää ajamatta.
- `--checks=*,-ssn,-captcha,-credit_card,-emails,-cvs_svn_users,-common_*, -backup_*,-backdoors`, joka poistaa käytöstä tarpeettomia testejä.
- `--scope-auto-redundant`, joka ottaa käyttöön samankaltaisten sivujen tunnistuslogiikan. Tämän avulla jätetään skannaamatta toisteiset sivut, jotka eivät luultavasti sisällä uutta sisältöä.
- `--scope-exclude-pattern=undefined`, joka estää skannauksen sivuilta, joiden URLissa on sana ”undefined”. Arachni teki jostain syystä paljon pyyntöjä virheellisiin sivuihin, joissa oleellinen parametri jäi arvoon ”undefined”, ja siten palautti aina sovelluksen virheilmoituksen. Tällä rajauksella saatiin estettyä nämä turhiksi tiedetyt pyynnöt.
- `--browser-cluster-pool-size=20`, joka kasvattaa Arachnin käyttämien selainins-tanssien määrää. Oletusarvo on 6. Koska testikoneella on käytössä runsaasti prosessoriytimiä, nopeuttaa rinnakkaistuksen lisääminen testiajon suoritusta.

Näillä Arachnin parametreilla ensimmäisen testattavan sovelluksen skannausajo lyheni noin neljään tuntiin, eikä sen aikana tullut enää aikakatkaisuvirheitä. Aikakatkaisuvirhei-

den syy jätettiin selvittämättä, koska selvitys olisi ollut työlästä ja koska optimoidussa ajossa niitä ei enää ilmennyt.

ZAPille ainoaksi tarpeelliseksi lisäparametriksi osoittautui `--exclude-rules 41,42,43,10027,10045,20017,40020,40021,40022,40024,40027,40033`, jolle valitut arvot poistavat käytöstä joukon *tahattoman lähdekoodin paljastuksia* (source code disclosure) etsiviä testejä, sekä sellaisiin tietokantatyyppeihin kohdistettuja testejä, joita ei kohdesovelluksessa käytetä. Tämän parametrin kanssa ZAP-ajo lyheni kuudesta tunnista noin kymmeneen minuuttiin.

Näillä optimoinneilla molemmat ajot saadaan ajettua yhden yön aikana, joten optimointien taso on riittävä tämän työn tarpeita varten.

6 Tulokset ja johtopäätökset

Skannausajoja ajettiin kahta sovellusta vasten. Molemmat sovellukset ovat keskikokoisia rekisterisovelluksia, jotka on toteutettu Kotlinilla/Javalla käyttäen Oikeiden Olioiden kehittämää Kanto-kehystä [Ruu04]. Kanto on white box -kehys, joka pyrkii helpottamaan ja nopeuttamaan tietokantaa käyttävien rekisterisovellusten kehitystä ja ylläpitoa toteuttamalla sovelluksille yhteisiä ominaisuuksia muun muassa käyttöliittymän ja käyttöoikeushallinnan osalta. Kantoa käyttävien sovellusten runkona on tietokannan relaatiomallia vastaava sovelluskonfiguraatio, jonka perusteella Kanto luo käyttöliittymän tietojen selaamiseen ja muokkaamiseen. Sovellusten käyttöliittymissä hyödynnetään hieman JavaScriptiä, mutta käyttöliittymän osat muodostetaan pääsääntöisesti palvelimella. Sovellus A käyttää MySQL-tietokantaa ja sovellus B käyttää PostgreSQL-tietokantaa.

		Skannauksen havainnot			
Skannausohjelmisto	**Ajon kesto**	**Korkea**	**Keskitaso**	**Matala**	**Info**
Arachni	55min	2	1	2	1
ZAP	20min	2	5	6	6

Taulukko 6.1: Sovellus A: Arachnin ja ZAPin löytämät havaintotyypit kahdesta sovelluksesta.

6.1 Arachni

Sovelluksesta A Arachni löysi kaksi korkean tason haavoittuvuutta: CSRF:n ja SQL-injektion. Molemmat näistä ovat vääriä positiivisia havaintoja. Keskitason havainto on salaamattomana lähetetty salasanakenttä. Tämäkään ei ole hyödyllinen havainto, sillä skannausympäristössä käytetään tarkoituksellisesti salaamatonta HTTP:tä yksinkertaisuuden ja paremman suorituskyvyn vuoksi. Skannausympäristössä käsitellään vain skannausohjelmistojen luomaa dataa, joten tämä ei aiheuta tietoturvariskiä.

Matalan tason havainnot ovat salasanakenttä automaattitäydennyksellä ja puuttuva `X-Frame-Options` -otsake, jotka molemmat ovat vääriä positiivisia. Salasanakentälle on määritelty `autocomplete="current-password"`-arvo, jota Arachni ei tunne, vaan neuvoo

käyttämään `off`-arvoa. `X-Frame-Options` -otsakkeen puute koskee vain sovelluksen juuressa olevaa uudelleenohjaussivua. Koska tällä sivulla ei ole mitään toiminnallisuutta, ei sitä voida käyttää clickjacking-höykkäykseen.

Info-tason havaintona on mielenkiintoinen vastaus, jossa Arachni on saanut sovellukselta HTTP 400 ja 500 -vastauksia yrittäessään antaa parametreille virheellisiä arvoja. Sovelluksen A skannaustulokset on esitetty taulukossa 6.1.

6.2 ZAP

ZAP puolestaan löysi sovelluksesta A CSRF- ja XSS-haavoittuvuuden. CSRF-tapaus on vastaava kuin Arachnin havainto. XSS-havainto toistuu yhteensä kahdeksassatoista eri paikassa, mutta manuaalisesti testattaessa ZAPin tarjoamat hyökkäysvektorit eivät tuota onnistunutta XSS-hyökkäystä. Sovelluksen näkökulmasta virheellisten syötteiden virheilmoitukset kuljetetaan käyttäjän näkyviin JavaScriptin avulla, jonka vuoksi ZAP luultavasti luulee hyökkäysyrityksen onnistuneen, vaikka arvojen käsittely onkin tehty oikein.

ZAPin keskitason havainnot olivat myös vääriä positiivisia:

- Järjestelmän virheilmoituksen paljastus: havainto löytyi dokumentaation osana olevasta merkkijonosta.
- Vain HTTP-protokolla on käytössä: testiympäristössä käytetään tarkoituksellisesti vain HTTP-protokollaa paremman suorituskyvyn vuoksi.
- Kokonaisluvun ylivuotovirhe: vääränmuotoisen tai liian ison kokonaisluvun syöttäminen aiheuttaa sovelluksessa virheen, mutta koska kyseessä on Java-sovellus, ei kokonaisluvun ylivuotoa pääse tapahtumaan.
- Haavoittuvainen JS-kirjasto: havainto kertoo jQuery 3.5.1:n olevan haavoittuvainen, mutta se on kirjaston uusin julkaistu versio, eikä siinä ole tunnettuja haavoittuvuuksia.
- `X-Frame-Options` -otsake asettamatta: havainto koskee vain sovellukseen upotettuja staattisia dokumentaatiosivuja, jotka ovat vain kirjautuneen käyttäjän nähtävissä.

ZAPin matalan tason havainnot:

- Järjestelmän virheilmoituksen paljastus: havainto löytyi dokumentaation osana olevasta merkkijonosta.
- Anti-CSRF-token puuttuu: ZAP ei löytänyt lomakkeista kenttää tunnetuilla CSRF-tokenin nimillä, sillä Kanto-sovelluksissa CSRF-tokenin nimi on sovelluskohtainen.
- Cookie Slack Detector: väittää, ettei sessioevästeen puuttumisella olisi joissain tapauksissa vaikutusta sivun sisältöön. Käsin varmistettaessa nämä tapaukset toimivat kuitenkin kuten kuuluukin.
- Eväste ilman SameSite-attribuuttia: havainto pitää paikkansa. Sovelluksen asettamista evästeistä puuttuu SameSite-attribuutti.
- Debug-viestien paljastus: havainto löytyi dokumentaation osana olevasta merkkijonosta.
- `X-Content-Type-Options` -otsake puuttuu: havainto pitää paikkansa. Sovellus ei aseta `X-Content-Type-Options` -otsaketta.

ZAPin info-tason havainnot:

- Cookie Slack Detector: väittää session mitätöityvän kielievästeen poistuessa. Manuaalisella testauksella ei saatu toistettua tätä havaintoa.
- Löyhästi määritellyt evästeet: sovellus ei määrittele asettamilleen evästeille täyttä osoitetta (FQDN, Fully Qualified Domain Name). Tämän ei pitäisi olla ongelma, sillä sovelluksien tuotantoympäristössä ei ole alidomaineja käytössä.
- Aikaleiman paljastus: havainnot ovat joko vääriä positiivisia, tai JavaScript-tiedostojen sisältä, joten ne eivät ole arkaluonteisia tietoja.
- User Agent Fuzzer: väittää sivun sisällön vaihtuvan selaimen kertoman User Agent -tiedon perusteella. Tämä on väärä positiivinen, sillä havainnon perusteena olevien sivujen sisältö vaihtuu jokaisella sivunlatauksella.
- User Controllable HTML Element Attribute: havaittujen parametrien arvojen sallitut arvot on ennalta määritelty sovelluskonfiguraatiossa, joten XSS-haavoittuvuus ei ole niiden tapauksessa mahdollista.
- User Controllable JavaScript Event: havaittujen parametrien käyttö XSS-hyökkäyksen toteuttamiseen ei manuaalisen testauksen perusteella ole mahdollista.

6.3 Toinen testattava sovellus

Seuraavaksi käydään läpi sekä Arachnin että ZAPin havainnot sovelluksesta B.

		Skannauksen havainnot			
Skannausohjelmisto	**Ajon kesto**	**Korkea**	**Keskitaso**	**Matala**	**Info**
Arachni	35min	1	0	1	3
ZAP	2h 20min	4	10	10	7

Taulukko 6.2: Sovellus B: Arachnin ja ZAPin löytämät havaintotyypit kahdesta sovelluksesta.

Sovelluksen B skannaustulokset ovat pitkälti vastaavat kuin sovelluksella A. Sovelluksesta B Arachni löysi havaintoja hieman vähemmän ja ZAP hieman enemmän. Sovelluksen B skannaustulokset on esitetty taulukossa 6.2.

Arachnin korkean tason havainto on CSRF, joka on väärä positiivinen vastaavasti kuin sovelluksen A kohdalla. Keskitason havaintoja ei löytynyt. Matalan tason havainto on puuttuva `X-Frame-Options` -otsake, vastaavasti kuin sovelluksen A kohdalla. Info-tason havainnot ovat HttpOnly-eväste, mielenkiintoinen vastaus ja ylimääräinen sallittu HTTP-metodi. HttpOnly-lipun puuttuminen sovelluksen asettamista evästeistä pitää paikkansa. Sovellus B käyttää vanhempaa Kanto-versiota, joka ei vielä tätä lippua evästeisiin aseta. Mielenkiintoinen vastaus aiheutuu Arachnin muodostamista epävalideista parametriarvoista. Ylimääräinen sallittu HTTP-metodi on HEAD, jonka sallimisen pitäisi olla turvallista.

ZAPin neljästä korkean tason havainnosta CSRF ja XSS olivat vastaavia vääriä positiivisia kuin sovelluksessa A. Toiset kaksi olivat polun läpikulkeminen ja SQL-injektio, jotka molemmat osoittautuivat manuaalitestauksessa vääriksi positiivisiksi.

Sovellusten ja skannaushavaintojen samankaltaisuuden vuoksi listataan seuraavaksi vain ne sovelluksen B skannauksessa löytyneet havainnot, joita ei löytynyt sovelluksen A skannauksessa.

ZAPin keskitason havainnot:

- Varmuuskopiotiedoston paljastuminen: havainnot koskivat JS- ja CSS-tiedostoja, joiden URLissa mukana olevaa istuntotunnistetietoa ZAP ei osannut tulkita oikein, vaan luuli jokaisen kokeilemansa varmuuskopiotiedoston olevan olemassa.

- Puskurin ylivuoto: jotkin liian pitkät syötteet johtavat sovellusvirheeseen, mutta koska kyseessä on Java-sovellus, ei varsinaista puskurin ylivuotoa ja siitä seuraavia tietoturvariskejä voi tapahtua.
- Hakemistolistaus: sovellukseen upotetuissa dokumentaatiokansioissa on hakemistojen sisällön listaus käytössä. Tämä on tarkoituksellista ja dokumentaatiokansiot ovat vain kirjautuneen käyttäjän nähtävissä.
- Merkkijonon muotoiluvirhe: havainnot eivät toistuneet manuaalisessa testauksessa. Tämä hyökkäystyyppi ei myöskään ole yhtä vakava Java-sovelluksissa, kuin mitä se voisi olla jollain muulla kielellä toteutetussa sovelluksessa.
- Reverse tabnabbing: dokumentaatiokansiossa olevissa SchemaSpyn [Sch20] luomissa sivuissa on `target='_blank'`-linkkejä, joilta puuttuu `noopener` ja `noreferrer` -avainsanat. Tätä ongelmaa ei ole enää SchemaSpyn uusimmassa versiossa.
- Istuntotunniste URLissa: istuntotunniste liitetään osaksi joidenkin tiedostojen URLia. Tämä ongelma on korjattu uudemmissa Kanto-kehyksen versioissa.

ZAPin matalan tason havainnot:

- Eväste ilman HttpOnly-lippua: havainto pitää paikkansa.
- IP-osoitteen paljastuminen: dokumentaatioissa on listattuna testiympäristöjen IP-osotteita, jotka eivät ole sovelluksen käyttäjille arkaluonteista tietoa.
- Referer paljastaa istuntotunnisteen: tämä on käytännössä sama havainto kuin keskitason istuntotunniste URLissa.
- Palvelin paljastaa versionsa Server-otsakkeessa: havainto koskee vain skannausten ajoympäristössä käytettyä kontitettua Nginx-palvelinta. Tuotantoympäristössä havainto ei toistu.

ZAPin ainoa uusi info-tason havainto oli evästeiden myrkytys. Se on väärä positiivinen, sillä kyseisen kielievästeen sallitut arvot on lueteltu sovelluksen konfiguraatiossa.

6.4 Johtopäätökset

Skannaustulokset olivat suurimmilta osin vääriä positiivisia molempien sovellusten kohdalla. Joukossa oli kuitenkin muutamia oikeitakin havaintoja, jotka on hyödyllistä ottaa huomioon sovellusten kehityksessä. Kummastakaan sovelluksesta ei löytynyt vakavia haavoittuvuuksia. Vaikka havainnoista suurin osa onkin vääriä positiivisia, tuovat skannausajot lisäarvoa sovellusten tietoturvan varmistamiseen. Skannausajoja käyttämällä on ainakin mahdollisuus löytää sovelluksista vakaviakin haavoittuvuuksia. Lisäksi varmistutaan siitä, etteivät mahdolliset hyökkääjät voi löytää sovelluksista helposti hyökkäyskohteita näitä samoja skannaustyökaluja käyttäen.

Sovellusten A ja B skannaustulokset olivat keskenään kohtalaisen samanlaisia. Tämä oli odotettavissa, sillä sovellukset on toteutettu käyttäen samaa Kanto-kehystä. Jos sovelluksista olisi löytynyt vakavampia haavoittuvuuksia, olisi havaintoja tullut luultavasti molemmista testattavaista sovelluksista.

Tietoturvaskannausten käyttöönotto uusissa sovelluksissa osoittautui kohtalaisen helpoksi. `.gitlab-ci.yml`-tiedostoon lisättävä ajokonfiguraatio toimii lähes samanlaisena sovellusten välillä. Sovelluskohtaista räätälöintiä tarvitaan mahdollisen sisäänkirjautumisen määrittämiseksi, sekä ajon nopeuden optimoimiseksi, joka tapahtuu rajaamalla turhat sovelluspolut ja testit pois käytöstä. Mikäli sovelluksen testiympäristöissä ei vielä hyödynnetä Dockeria, suurempi työ saattaa olla sovelluksen saattamisessa sellaiseen kuntoon, että siitä voidaan käynnistää uusi Dockeroitu instanssi GitLab CI -ajon aikana.

Tämän tutkimuksen aikarajoitteiden ja testattavien sovellusten vaihtelevan kehitystahdin vuoksi skannaustuloksia ei ehditty seuraamaan sovelluskehityksen edistyessä. Kiinnostavaa olisikin seurata skannaustulosten kehitystä pidemmällä aikavälillä, jotta nähtäisiin, kuinka arvokkaita skannaustulokset voivat olla.

Seuraava askel skannausajojen parantamiseen voisi olla hälytysten lisääminen uusien skannaushavaintojen ilmaantuessa, jotta tietoturvaregressiot varmasti havaittaisiin mahdollisimman pian.

Toinen mahdollinen kehityskohde voisi olla sovelluskohtaisten integraatiotestien hyödyntäminen tietoturvaskannauksissa. Skannausohjelmistojen tekemän automaattisen sovelluksen toimintojen etsimisen ja läpikäynnin lisäksi integraatiotestien avulla voitaisiin paremmin näyttää skannausohjelmistoille mitä toimintoja sovelluksessa on ja miten niitä kuuluu käyttää. Tämä voisi parantaa skannauksen kattavuutta.

7 Yhteenveto

Tässä työssä käytiin läpi web-sovellusten, testauksen ja riskienhallinnan käsitteistöä ja historiaa, eriteltiin web-sovellusten tietoturvariskejä OWASP Top 10 -listauksen avulla ja perehdyttiin web-sovellusten tietoturvan testaukseen automaattisten skannausohjelmistojen avulla.

Ilmaisten ja maksullisten skannausohjelmistojen vertailun perusteella päädyttiin käyttämään ilmaisia skannausohjelmistoja ja Chenin vertailun perusteella päädyttiin käyttämään Arachnia ja ZAPia [Che14]. Skannausohjelmistot eivät soveltuneet SPA-sovellusten testaamiseen, joten keskityttiin perinteisten web-sovellusten testaamiseen. Esiteltiin käytettäväksi valitut skannausohjelmistot lyhyesti.

Arachnin ja ZAPin automaattiset skannausajot liitettiin osaksi kahden testattavan sovelluksen GitLab CI -käännösputkea. Skannausajojen tehokkuutta parannettiin optimoimalla skannausohjelmistojen käynnistysparametreja. Arachnin ja ZAPin skannaustulokset kahdesta kohdesovelluksesta käytiin läpi.

Tuloksista suurin osa oli vääriä positiivisia, mutta joukossa oli myös kiinnostavia havaintoja. Skannausajojen käyttöönotto osaksi regressiotestausta havaittiin hyödylliseksi. Tietoturvaskannauksen käyttöönotto uusissa sovelluksissa todettiin helpoksi.

Kehityskohteita olisi hälytysten lisäämisessä skannausajoihin ja integraatiotestien hyödyntämisessä osana skannausajoja. Jatkotutkimusta voisi tehdä seuraamalla skannausajojen tuloksia pidemmällä aikavälillä sovelluskehityksen aikana.

Lähteet

[Abe14] R. Abela. *An Automated Scanner That Finds All OWASP Top 10 Security Flaws, Really?* https://www.netsparker.com/blog/web-security/owasp-top-10-web-security-scanner/. [Online; haettu 27.11.2020]. 2014.

[Ang20] Angular. *Angular*. https://angular.io/. [Online; haettu 27.11.2020]. 2020.

[Ben20a] S. Bennetts. *OWASP Zed Attack Proxy - Docker*. https://www.zaproxy.org/docs/docker/. [Online; haettu 27.11.2020]. 2020.

[Ben20b] S. Bennetts. *Zed Attack Proxy*. https://www.zaproxy.org/. [Online; haettu 27.11.2020]. 2020.

[Che14] S. Chen. *WAVSEP 2013/2014 Score Chart: The Web Application Vulnerability Scanners Benchmark*. http://sectooladdict.blogspot.fi/2014/02/wavsep-web-application-scanner.html. [Online; haettu 27.11.2020]. 2014.

[Che15] S. Chen. *Price and Feature Comparison of Web Application Scanners*. http://www.sectoolmarket.com/price-and-feature-comparison-of-web-application-scanners-opensource-list.html. [Online; haettu 27.11.2020]. 2015.

[Doc20] Docker. *Docker*. https://www.docker.com/. [Online; haettu 27.11.2020]. 2020.

[DRP99] E. Dustin, J. Rashka ja J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[Eur19] Eurostat. *Internet use by individuals*. https://ec.europa.eu/eurostat/web/products-datasets/product?code=tin00028. [Online; haettu 27.11.2020]. 2019.

[FF06] M. Fowler ja M. Foemmel. *Continuous integration*. https://martinfowler.com/articles/continuousIntegration.html. [Online; haettu 27.11.2020]. 2006.

[Fos18] E. Foster. ”Testing Web Application Security Scanners against a Web 2.0 Vulnerable Web Application”. Teoksessa: 2018.

[Git20a] GitLab. *GitLab Docker Build*. https://docs.gitlab.com/ee/ci/docker/using_docker_build.html. [Online; haettu 27.11.2020]. 2020.

[Git20b] GitLab. *GitLab Pipeline schedules*. https://docs.gitlab.com/ee/ci/pipelines/schedules.html. [Online; haettu 27.11.2020]. 2020.

[Git20c] GitLab. *GitLab Runner*. https://docs.gitlab.com/runner/. [Online; haettu 27.11.2020]. 2020.

[Git20d] GitLab. *GitLab Runner Executors*. https://docs.gitlab.com/runner/executors/README.html. [Online; haettu 27.11.2020]. 2020.

[Git20e] Gitlab. *GitLab CI*. https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/. [Online; haettu 27.11.2020]. 2020.

[Hom08] D. of Homeland Security (US). *DHS Risk Lexicon*. Department of Homeland Security (US), 2008.

[Idr+17] S. E. Idrissi, N. Berbiche, F. Guerouate ja M. Sbihi. "Performance Evaluation of Web Application Security Scanners for Prevention and Protection against Vulnerabilities". *International Journal of Applied Engineering Research* 12 (2017).

[Int18] International Organization for Standardization. *ISO 27000: Information technology — Security techniques — Information security management systems — Overview and vocabulary*. Standard. 2018.

[KQS18] T. Khairallah, M. Qasaimeh ja A. Shamlawi. "Black Box Evaluation of Web Application Scanners: Standards Mapping Approach". *Journal of Theoretical and Applied Information Technology* 96.14 (2018).

[Kup20] L. Kuppan. *IronWASP*. https://github.com/Lavakumar/IronWASP/. [Online; haettu 27.11.2020]. 2020.

[Las20a] T. Laskos. *Arachni Web Application Security Scanner Framework*. https://www.arachni-scanner.com/. [Online; haettu 27.11.2020]. 2020.

[Las20b] T. Laskos. *Arachni: Crawl coverage and vulnerability detection*. http://www.arachni-scanner.com/features/framework/crawl-coverage-vulnerability-detection/. [Online; haettu 27.11.2020]. 2020.

[Man11] S. Mansfield-Devine. "Hacktivism: assessing the damage". *Network Security* 2011.8 (2011), s. 5–13.

[McQ14] K. McQuade. ”Open Source Web Vulnerability Scanners: The Cost Effective Choice?” Teoksessa: *Proceedings of the Conference for Information Systems Applied Research ISSN*. Vol. 2167. 2014, s. 1508.

[MM14] A. Muller ja M. Meucci. *Testing Guide 4.0*. The OWASP Foundation, 2014.

[MP13] M. Mikowski ja J. Powell. *Single page web applications: JavaScript end-to-end*. Manning Publications Co., 2013.

[MSB11] G. J. Myers, C. Sandler ja T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[Oli20] O. Oliot. *Oikeat Oliot*. https://oikeatoliot.fi/. [Online; haettu 27.11.2020]. 2020.

[OWA13] OWASP. *Top 10 - 2013, The Ten Most Critical Web Application Security Risks*. https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf. 2013.

[OWA17] OWASP. *Top 10 - 2017, The Ten Most Critical Web Application Security Risks*. https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010-2017%20(en).pdf. 2017.

[OWA20a] OWASP. *OWASP Juice Shop*. https://owasp.org/www-project-juice-shop/. [Online; haettu 27.11.2020]. 2020.

[OWA20b] OWASP. *OWASP Top 10 Github*. https://github.com/OWASP/Top10. [Online; haettu 27.11.2020]. 2020.

[Por20] PortSwigger. *Burp Suite*. https://portswigger.net/burp. [Online; haettu 27.11.2020]. 2020.

[Ruu04] J. Ruuttunen. ”A product line architecture for web-based data management systems”. Diplomityö. Teknillinen korkeakoulu, 2004.

[Sch20] SchemaSpy. *SchemaSpy*. http://schemaspy.org. [Online; haettu 27.11.2020]. 2020.

[SE14] F. A. Saeed ja E. A. Elgabar. ”Assessment of open source web application security scanners”. *Journal of Theoretical and Applied Information Technology* 61.2 (2014).

[Sub20] Subgraph. *Vega*. https://subgraph.com/vega/. [Online; haettu 27.11.2020]. 2020.