Automatisoitu tietoturvatestauksen toteutus rajapintojen kehitykseen CI/CD -tuotantoympäristössä

Diplomityö
Turun yliopisto
Tietotekniikan laitos
Kyberturvallisuusteknologia
2024
Iikka Luoma-aho

Tarkastajat:
Petri Sainio
Seppo Virtanen

TURUN YLIOPISTO
Tietotekniikan laitos

Iikka Luoma-aho: Automatisoitu tietoturvatestauksen toteutus rajapintojen kehitykseen CI/CD -tuotantoympäristössä

Diplomityö, 57 s.
Kyberturvallisuusteknologia
Kesäkuu 2024

Tässä diplomityössä tarkastellaan automatisoidun tietoturvatestauksen toteutusta rajapintojen kehityksessä jatkuvan integroinnin ja jatkuvan toimittamisen (CI/CD) tuotantoympäristössä. Työn keskeisenä tavoitteena on ollut arvioida menetelmiä, jotka integroivat tietoturvatestauksen osaksi jatkuvaa ohjelmistokehitysprosessia. Näin pyritään parantamaan ohjelmistojen turvallisuutta ja vähentämään haavoittuvuuksien esiintymistä tuotantoympäristöissä. Erityisesti keskitytään avoimen lähdekoodin staattisiin ja dynaamisiin tietoturvatestausmenetelmiin sekä interaktiiviseen tietoturvatestaukseen, jotka mahdollistavat syvällisen haavoittuvuusanalyysin suorittamisen ohjelmistokehityksen eri vaiheissa.

Tämä tutkimus keskittyy tietoturvan tarkasteluun erityisesti ohjelmoitavien rajapintojen (API) näkökulmasta, mikä korostaa CI/CD-ympäristön merkitystä turvallisen ohjelmistokehityksen mahdollistajana. Tutkimuksessa suoritetaan analyysi eri tietoturvatestausmenetelmien soveltuvuudesta jatkuvaan integrointiin ja jatkuvaan toimittamiseen (CI/CD) perustuvaan tuotantoympäristöön, niiden tehokkuudesta tunnistaa haavoittuvuuksia sekä niiden integraation vaikutuksesta kehitysprosessiin. Lisäksi työssä verrataan avoimen lähdekoodin ja kaupallisten tietoturvatestaustyökalujen suorituskykyä ja soveltuvuutta rajapintakehitykseen. Järjestelmien käytettävyyttä kehitysprosessin osana arvioidaan kehittäjien näkökulmasta.

Tutkimuksen keskeiset tulokset viittaavat siihen, että automatisoidun tietoturvatestauksen käyttöönotto CI/CD-tuotantoympäristössä voi olennaisesti lisätä ohjelmistotuotteiden turvallisuutta. Tutkimuksessa tunnistetaan myös useita haasteita ja rajoitteita, jotka liittyvät automatisoitujen testausmenetelmien käyttöönottoon ja soveltamiseen käytännön tilanteissa. Tutkimuksen perusteella esitetään suosituksia tietoturvatestauksen kehittämiseksi sekä toimenpide-ehdotuksia, jotka tukevat organisaatioita tietoturvatestauksen integroimisessa osaksi jatkuvaan integrointiin ja jatkuvaan toimittamiseen perustuvaa tuotantoympäristöä. Tutkimuksessa suositellaan käytettävän useampaa erityyppistä testausmenetelmää, jolloin voidaan saavuttaa paras kattavuus.

Asiasanat: automatisoitu tietoturvatestaus, ohjelmoitava rajapinnan tietoturvan arviointi, CI/CD-putken tietoturvaimplementaatio

UNIVERSITY OF TURKU
Department of Computing

Iikka Luoma-aho: Automatisoitu tietoturvatestauksen toteutus rajapintojen kehitykseen CI/CD -tuotantoympäristössä

Master of Science Thesis, 57 p.
Cyber security
June 2024

In this thesis, the implementation of automated security testing in the development of interfaces within a continuous integration and continuous delivery (CI/CD) production environment is examined. The primary objective of this work has been to evaluate methods that integrate security testing into the continuous software development process. This aims to enhance the security of software products and reduce the occurrence of vulnerabilities in production environments. The focus is particularly on open-source static and dynamic security testing methods, as well as interactive security testing, which enable comprehensive vulnerability analysis at various stages of software development.

This research focuses on the examination of security from the perspective of programmable interfaces (APIs), highlighting the importance of the CI/CD environment as an enabler of secure software development. The study conducts an analysis of the suitability of different security testing methods for continuous integration and continuous delivery (CI/CD) based production environments, their effectiveness in identifying vulnerabilities, and their impact on the development process. Additionally, the performance and appropriateness of open-source versus commercial security testing tools for interface development are compared. The usability of systems as part of the development process is assessed from the developers' perspective.

The key findings of the research suggest that the adoption of automated security testing in a CI/CD production environment can significantly increase the safety of software products. The study also identifies several challenges and limitations associated with the adoption and application of automated testing methods in practical situations. Based on the research, recommendations are made for improving security testing and action proposals that support organizations in integrating security testing into a continuous integration and delivery-based production environment. The study recommends the use of multiple types of testing methods to achieve the best coverage.

Keywords: Automated security testing, API Security assesment, CI/CD security implementation

Sisällys

1 Johdanto

Yrityksien ohjelmistokehityksen suureksi tarpeeksi on noussut jatkuva uusien ja vanhojen palveluiden ja sovellusten kehittäminen. Tämä kehitystyö toteutetaan usein rajapintojen, eli sovellusohjelmointirajapintojen (API), avulla. Rajapinnat mahdollistavat kommunikoinnin eri ohjelmien ja tietokantojen välillä, ja niiden avulla voidaan luoda uusia toiminnallisuuksia ohjelmistoihin. Yksi merkittävä etu rajapintaohjelmistoissa on tunnistautumismekanismien avulla tapahtuva mukautettavuus. Se mahdollistaa yhden palvelun tarjoamisen useammalle käyttäjäryhmälle, joiden saaman informaation määrää voidaan rajoittaa käyttäjäoikeuksien perusteella. Sama rajapinta voi esimerkiksi palauttaa eri tietoja eri käyttäjille: toiselle käyttäjälle vain perustietoja palvelusta ja toiselle lisätietoja perustietojen lisäksi [1], [2].

Rajapintojen käyttö on yleistynyt monilla liiketoimintaaloilla. Esimerkiksi pankit tarjoavat rajapintoja verkkokaupoille verkkomaksujen suorittamiseen. Verkkokauppa voi hyödyntää tätä rajapintaa asiakkaan maksun prosessointiin, jolloin erillistä paperilaskua ei tarvita. Tämä mahdollistaa verkkokaupalle asiakkaiden vakuuttamisen maksutapahtuman turvallisuudesta. Pankki puolestaan voi taata asiakkaalleen turvallisen maksutapahtuman tarjoamalla kolmannen osapuolen toimijalle mahdollisuuden laskuttaa pankin asiakkaita rajapinnan avulla. Tällä tavoin sekä pankki, verkkokauppa että asiakas hyötyvät rajapinnasta [2].

Tietoturva on keskeinen huolenaihe nykypäivän digitaalisessa ympäristössä, ja sen merkitys korostuu jatkuvasti ohjelmistokehityksen kontekstissa. Ketterän kehi-

tyksen paradigma on nopeuttanut ohjelmistokehitystä ja tuonut mukanaan merkittävän määrän automaatiota. Tämä kehitys on tuonut esille automatisoinnin hyödyt, erityisesti tietoturvan testauksessa. Stefinko et al. ovat esittäneet tutkimuksessaan useita etuja automatisoidulle tietoturvatestaukselle [3]. Vaikka avoimen lähdekoodin tukemia automatisoituja tietoturvan testausjärjestelmiä on saatavilla, niiden välisiä tieteellisiä vertailuja on vaikea löytää. Lisäksi useat tutkimukset ovat puolueellisia ja keskittyvät yksittäisen järjestelmän tarkasteluun. Esimerkiksi *Synopsis*-yritys on julkaissut tutkimuksia omasta *Seeker*-järjestelmästään [4]. Mburano ja Si ovat tehneet vertailevan tutkimuksen [5], jossa useita järjestelmiä testattiin samoilla kriteereillä. Tässä tutkimuksessa keskityttiin kuitenkin järjestelmien kykyyn havaita yleisiä uhkia, eikä otettu kantaa siihen, miten järjestelmät toimisivat automatisoidussa ympäristössä.

1.1 Tutkimuksen tarkoitus

Aihetta koskevan puolueettoman tiedon vähyydestä nousi into selvittää, millaisia avoimen lähdekoodin järjestelmiä on tällä hetkellä olemassa. Erityisesti kiinnostuksen kohteena on tutkia, miten nämä järjestelmät tunnistavat rajapintojen haavoittuvuuksia. Tutkimuksen tarkoituksena on perehtyä tarkemmin rajapintojen tietoturvallisuuden automatisointiin ja ehdottaa millainen automaatiota tukeva testausjärjestelmä olisi. Tavoitteena on tarjota kattava kuvaus ratkaisuista ja arvoida niiden soveltuvuutta etenkin rajapintapohjaisten sovellusten tietoturvan automaattiseen arviointiin.

Ensimmäiseksi lopulliseksi tutkimuskysymykseksi muodostui seuraava: Millaisia avoimen lähdekoodin automatisoituja tietoturvajärjestelmiä on olemassa ja miten ne soveltuvat rajapintojen kehittämiseen? Tätä kuvausta tullaan käyttämään implementaatio ehdotuksena yrityksen automaattisen tietoturvatestauksen päivityksessä ketterää kehitystä tukevaksi. Toiseksi tutkimuskysymykseksi muodoistui kysymys

siitä, miten avoimeen lähdekoodiin perustuvat automaattisen tietoturvatestauksen järjestelmät soveltuvat rajapintojen haavoittuvuuksien tunnistamiseen.

1.2 Tutkimuksen toteuttaminen

Tutkimus on toteutettu kirjallisuuskatsauksen ja käytännön testauksen avulla. Kirjallisuuskatsauksen osuudessa selvitettiin, mitä automatisoidut tietoturvatestit ovat ja mitä haavoittuvuuksia rajapintaohjelmistoilla on. Kirjallisuuskatsauksen aineistot perustuvat kansainvälisiin tieteellisiin artikkeleihin, julkaisukokoelmiin ja seminaareihin. Käytännön testauksen osuudessa keskityttiin avoimen lähdekoodin ratkaisujen esittelyyn ja testaamiseen. Käytännön testauksen järjestelmät on valittu niiden ohjelmistokehitysyhteisössä saaman laajan käytön ja tunnustuksen perusteella. Testauksen tavoitteena on arvioida sekä näiden järjestelmien kykyä havaita haavoittuvuuksia että niiden soveltuvuutta automatisoituun ohjelmistokehitykseen, erityisesti CI/CD-prosessin kontekstissa.

1.3 Tutkielman rakenne

Tutkielma alkaa johdannosta, jossa esitellään aihe. Toisessa luvussa käydään läpi, mitä on tietoturvatestaus, miten se on osa CI/CD-prosessia ja millaisia erilaisia automatisoituja tietoturvatestausjärjestelmiä on olemassa. Kolmannessa luvussa esitellään tarkemmin, mitä ohjelmoitavat rajapinnat ovat. Luvun alussa käsitellään ohjelmoitavien rajapintojen toimintaperiaatteita, ja luvun lopussa esitellään suurimpia rajapintajärjestelmien tietoturvahaavoittuvuuksia. Neljännessä luvussa tutkitaan eri tietoturvajärjestelmäarkkitehtuureihin perustuvia avoimen lähdekoodin järjestelmiä. Viidennessä luvussa käydään läpi, miten testaus tässä tutkimuksessa toteutettiin. Kuudennessa luvussa esitellään tuloksia. Viimeisessä luvussa pohditaan tuloksia ja tutkimuksen luotettavuutta sekä käydään läpi jatkotutkimuksen.

2 Automatisoitu tietoturvatestaus

Tämä luku keskittyy tietoturvatestauksen määritelmään, sen kehitysprosessiin integrointiin sekä sen eri muotoihin. Erityisesti luku syventyy tietoturvatestauksen automatisoituihin järjestelmätekniikoihin.

2.1 Tietoturvatestaus

Tietoturvatestaus on prosessi, joka tutkii tietoturvan vaatimuksia ohjelmiston suunnittelussa ja toteutuksessa. Sen tavoitteena on varmistaa ohjelmiston tietoturvallisuus eli suojata se tunnetuilta haavoittuvuuksilta ja tietoturvauhilta. Haavoittuvuus on järjestelmän sisäinen tekijä, joka heikentää järjestelmän turvallisuuden. Haavoittuvuus voi johtua esimerkiksi järjestelmän laitteistosta tai ohjelmistosta. Tietoturvauhka on ulkoinen tekijä tai tapahtuma, joka vaarantaa järjestelmän luottamuksellisuuden, eheyden, saatavuuden tai yksityisyyden. Tietoturvauhissa hyödynnetään järjestelmän haavoittuvuuksia. Tietoturvavaatimukset ja -määrittelyt ovat keskeisiä, sillä ne estävät tietoturvauhkien pääsyn ohjelmistoon [6].

Tietoturvatestauksen voi jakaa kahteen osa-alueeseen: turvallisuuden toiminnallisuuden testaukseen ja turvallisuusuhkien testaukseen. Turvallisuuden toiminnallisuuden testauksessa keskitytään tietoturvallisuuden ominaisuuksien toteutukseen ja tarkastellaan, onko tarvittavat turvallisuusmäärittelyt otettu huomioon ohjelmiston toteutuksessa. Tavoitteena on varmistaa, että turvallisuusmäärittelyt vastaavat tietoturvavaatimuksia. Haavoittuvuuksien testauksessa puolestaan tarkastellaan oh-

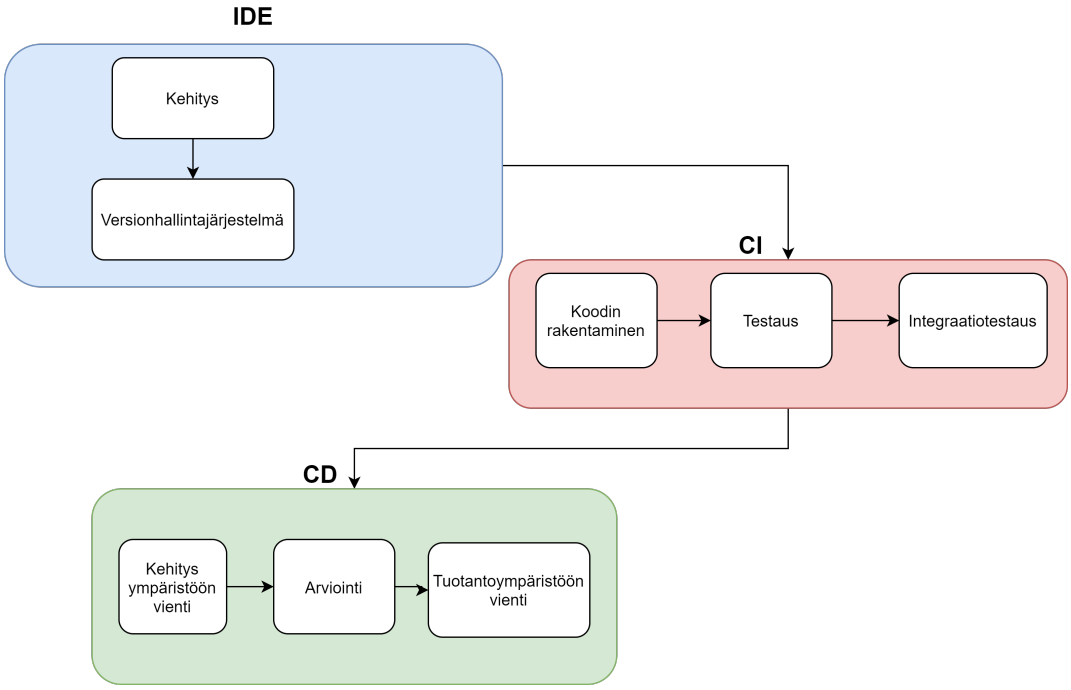
jelmistoa hyökkääjän näkökulmasta etsien mahdollisia haavoittuvuuksia. Haavoittuvuudella viitataan järjestelmän väärinkäytön mahdollistavaan virheeseen ohjelmiston suunnittelussa tai toteutuksessa. Yleisesti tunnettuja haavoittuvuuksia seurataan ja kirjataan tietokantoihin, jotka ovat hyödyllisiä tietoturvatestauksessa. Verkosta löytyy sekä avoimia että kaupallisia uhkatietokantoja, joista saa tietoa uhista, niiden kuvauksista ja esimerkeistä. [7]

Yksi tietoturvatestauksen keskeisistä menetelmistä on mallianalyysi. Mallianalyysit ovat hyödyllisiä sekä turvallisuuden toiminnallisuuden että turvallisuusuhkien testaamisessa. Toiminnallisuuden analyysissa malli keskittyy järjestelmän toiminnallisuuksien tutkimiseen, erityisesti siihen, miten järjestelmän operaatiot ja funktiot käsittelevät dataa. Malli arvioi, aiheuttavatko toteutukset riskejä ja millaisia mahdolliset riskit ovat. Analyysi voi ulottua käyttäjän syötteiden käsittelystä datanmuuntamisoperaatioihin. Uhkapohjaisissa mallianalyyseissa keskitytään riskianalyysiin ja yleisesti tunnettujen uhkien testaamiseen [8]. Uhkien testaamisessa hyödynnetään *CVE (Common Vulnerabilities and Exposures)* -tietokantoja. Tavoitteena uhkapohjaisissa mallianalyyseissa on tunnistaa ja korjata järjestelmän haavoittuvuudet. Järjestelmän haavoittuvuuksien tunnistaminen voi perustua koodin tutkimiseen tai järjestelmän testaamiseen simuloimalla erilaisia tunnettuja uhkia. Riskianalyysi auttaa tunnistamaan uhkia, jotka liittyvät järjestelmän ylläpitoon tai hallintaan. Testaukset voidaan suorittaa käyttäen erilaisia manuaalisia tai automatisoituja työkaluja. [9]

2.2 Tietoturvatestaus CI/CD-prosessissa

CI/CD (Continuous Integration/Continuous Delivery) on modernin ohjelmistokehityksen prosessi, joka yhdistää jatkuvan integraation ja jatkuvan tuotantoon viennin yhdeksi prosessiksi. Lisäksi se sisältää integroidut kehitysympäristöt (IDE, *Integrated Development Environment*). Tämä kokonaisuus tarkoittaa, että ohjelmiston

kehitys tapahtuu jatkuvasti ja uusimmat päivitykset ovat välittömästi saatavilla tuotantoympäristössä. Kuten kuva 2.1 osoittaa, testaus on olennainen osa tätä prosessia: se on keskeisessä osassa varmistamassa, että tuotantoon ei pääse keskeneräisiä toiminnallisuuksia tai haavoittuvuuksia. CI/CD hyödyntää automaatiota erityisesti tuotantoon viemisen ja testauksen vaiheissa, mikä mahdollistaa ketterän ohjelmistokehityksen ja mahdollistaa nopeamman reagoinnin tuotantoympäristössä ilmeneviin haasteisiin ja tarpeisiin.

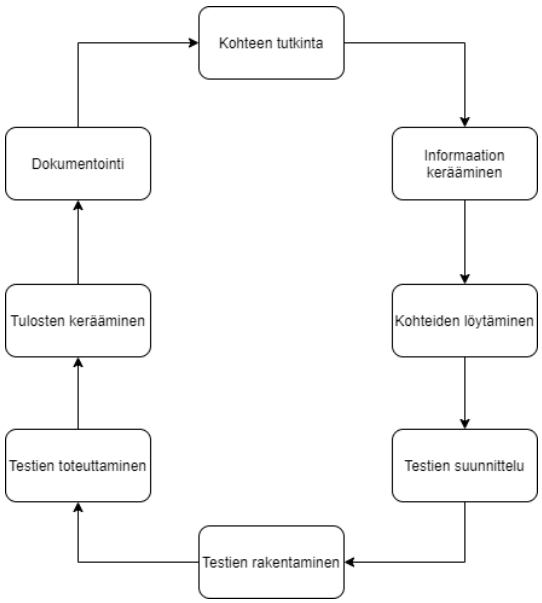


Kuva 2.1: CI/CD-prosessin toiminta eri vaiheissa [10].

2.3 Manuaalinen tietoturvatestaus

Manuaalinen tietoturvatestaus perustuu valmiiden työkalujen tai käsin kirjoitettujen hyökkäyksien tekemiseen kohdejärjestelmää vastaan. Manuaalinen tietoturvatestaus pyrkii löytämään haavoittuvuuksia järjestelmästä. Testaus vaatii testaajalta laajaa osaamista tietoturva-alalta sekä erilaisista työkaluista ja ohjelmointikielistä.

Testaajan on oltava perehtynyt uusimpiin tietoturvauhkiin ja seurattava aktiivisesti muuttuvaa alaa. Lisäksi manuaalinen testaus vaatii paljon resursseja, kuten aikaa suorittaa analyseeja ja tutkia tuloksia. [3]



Kuva 2.2: Prosessikuvaus manuaalisen tietoturvatestauksen vaiheista

Manuaalinen testaus perustuu sovellettuihin metodologioihin. Nämä metodologiat tarjoavat testausprosessille rakenteen ja johdonmukaisuuden, mikä mahdollistaa testauksen toistettavuuden ja vertailukelpoisuuden.Tunnettuja tietoturvatestauksessa hyödynnettäviä metodologioita ovat *OSSTMM (Open Source Security Testing Methodology Manual)*, *OWASP (Open Web Application Security Project)*, *NIST (National Institute of Standards and Technology)* ja *PTES (Penetration Testing Execution Standard)*. Kuvasta 2.2 nähdään yleinen kuvaus testausprosessista. Haavoittuvuuksien löytäminen ja dokumentointi ovat avainasemassa järjestelmän turvallisuuden parantamisessa. Haavoittuvuuksien löytäminen on vaihe, jossa testaaja ajaa testit järjestelmää vasten ja pyrkii todentamaan haavoittuvuuksia järjestelmästä. Dokumentoinnissa kirjataan muistiin sekä tietoa löydetyistä haavoittuvuuksista että suosituksia turvallisuuden parantamiseen. Metodologieiden käyttö tukee kehittäjiä ja tietoturvatiimejä pyrittäessä systemaattisesti vahvistamaan oh-

jelmistojen ja järjestelmien turvallisuutta. [11], [12]

Verrattuna automaattisiin testeihin manuaalinen tietoturvatestaus tarjoaa kohdennettua tutkimista. Testaajat suorittavat käsintehtyjä testejä, joiden tarkoituksena on tunnistaa järjestelmässä olevat tietoturvahaavoittuvuudet. Käsintehdyt testit mahdollistavat testaajien kohdennetun syventymisen järjestelmän erityispiirteisiin ja potentiaalisiin heikkouksiin, jotka automatisoidut testit saattavat ohittaa. Manuaalinen tietoturvatestaus vaatii kuitenkin paljon resursseja. Testaajilta vaaditaan aikaa, syvää osaamista tietoturvauhista ja kyky ymmärtää monimutkaisia järjestelmiä. Manuaalinen testaus hidastaa kehitysprosessia, ja jatkuvan integraation ja jatkuvan toimituksen prosessissa se voidaan kokea hidastavaksi tekijäksi [3].

Automatisoidut tietoturvatestausohjelmat ovat kehitetty sen takia, että manuaalinen testaus on resurssitehotonta. Automatiikan avulla kohdejärjestelmälle pystytään ajamaan suurempi määrä testejä. Automatisoitu tieturvatestaus antaa mahdollisuuden nopeammalle testaukselle, sillä se voidaan toteuttaa suoraan kehitysjärjestelmiin ja ajaa koodille samaan aikaan kuin muut ohjelmistotestaukset. Lisäksi testien määrä on moninkertainen verrattuna manuaaliseen testaukseen, sillä valmiita tietokantoja käyttäen voidaan ajaa useamman eri uhan testi. Ongelmaksi automaattisessa testauksessa nousevat testien luotettavuus ja niiden rajallinen mahdollisuus ymmärtää koodia . Automaatio tunnistaa ja tekee testit yleisten esimerkkien pohjalta. Tästä ongelmaksi voi nousta järjestelmää kohtaan suunnitellut hyökkäykset, joissa hyödynnetään järjestelmän tuntemusta. Tällaisia uhkia automaattiset järjestelmät eivät pysty testaamaan, sillä ne eivät ole tietoisia testatusta järjestelmästä.

2.4 Automatisoitu tietoturvatestaus

Automatisoidun tietoturvatestauksen ydin on käyttää samoja välineitä kuin manuaalisessa testauksessa Eroavaisuus ilmenee testaustekniikoissa. Automatisoidun tietoturvatestauksen prosessissa sovelletaan ohjelmistoautomaatiota kohdejärjestelmän

testaamiseen ja tietoturvariskien todentamiseen. Automaatio ohjelmoi ja suorittaa koodisarjoja tutkiakseen niiden potentiaalista uhkaa tietoturvalle. Nämä koodisarjat ovat tyypillisesti geneerisiä ja pohjautuvat tunnettuihin turvallisuusriskeihin, esimerkiksi CVE-tietokantoihin. Testien suorittamisen jälkeen automaatiojärjestelmä raportoi löydökset. Automaattisen tietoturvatestauksen keskeisiä hyötyjä on kyky suorittaa laaja kirjo testejä kohdejärjestelmässä ja tuottaa tulokset tehokkaasti. [3]

Automaatio nopeuttaa tietoturvatestauksen prosessia mahdollistamalla kattavamman testien kirjon kohdejärjestelmässä. Sen avulla testauksen voi yhdistää suoraan kehitysprosessiin ja suorittaa samanaikaisesti muiden testien kanssa. Automaattisen tietoturvan testauksen hyötyihin kuuluu kyky suorittaa useampia testejä erilaisia uhkaskenaarioita vastaan käyttäen valmiita tietokantoja, mikä ylittää manuaalisen testauksen rajat. Automaation haasteisiin kuuluu puolestaan testien luotettavuuden ja koodin ymmärryksen rajallisuus. Automaatio perustuu yleisiin malleihin, mikä voi aiheuttaa haasteita tunnistaa järjestelmään kohdistuvia erityisiä hyökkäyksiä, jotka edellyttävät järjestelmän syvällistä tuntemusta. Tällaiset spesifiset uhat saattavat jäädä automatisoidun testauksen ulkopuolelle.

Automaattisen tietoturvatestauksen ohjelmia on useita tyyppejä ja monet niistä sisältävät komponentteja automatisoitua testausta varten. Nämä ohjelmat ovat saatavilla sekä kaupallisin että avoimen lähdekoodin lisenssein. Ne tarjoavat erilaisia testausmahdollisuuksia ja jotkut on suunnattu erityistyyppisiin uhkiin tai laajempiin testauskokonaisuuksiin. Testausohjelmat voidaan luokitella kolmeen pääryhmään niiden toimintatavan mukaan: *SAST (Static Application Security Testing)*, *DAST (Dynamic Application Security Testing)* ja *IAST (Interactive Application Security Testing)*, joista jokainen hyödyntää erilaista lähestymistapaa tietoturvan testauksessa. Lisäksi on olemassa *RASP (Runtime Application Self-Protection)* , joka tarjoaa ajonaikaista tietoturvaa ja esitellään yksityiskohtaisemmin myöhemmin. [13]

2.4.1 SAST-järjestelmä

Staattista koodianalyysia hyödynnetään ohjelmistotuotannossa laajalti tuotetun koodin laadun varmistamiseksi. Sillä mahdollistetaan myös koodin samanlaisuus, mikä tukee koodin luettavuutta. SAST-järjestelmät toteuttavat staattista analyysia havaitakseen tietoturvahaavoittuvuuksia ohjelmiston lähdekoodista ja käyttävät hyväkseen tietoturvahaavoittuvuustietokantoja. Analyysi kattaa lähdekoodin komponentit ja tuottaa kategorisoinnin esimerkiksi haavoittuvuuden tyypin tai vakavuuden mukaan. Järjestelmän vahvuutena pidetään sen käyttöönoton helppoutta, sillä se voidaan integroida sujuvasti kehittäjän editoriin tai ohjelmistohallintajärjestelmiin. SAST-järjestelmän etuna on myös sen kyky tunnistaa pelkän koodin perusteella ihmiseen verrattuna useampia haavoittuvuuksia [14].

SAST-järjestelmän merkittävänä haasteena on sen tuottamien ilmoitusten ja huomioiden suuri määrä. Järjestelmän analytiikan vuoksi se tuottaa runsaasti informaatiota koodista, mikä voi olla haastavaa kehittäjälle, joka pyrkii ymmärtämään varoitusten merkitykset ja priorisoimaan niistä kriittisimmät. Johnson et al. ovat nostaneet esille, että väärien positiivisten hälytysten suuri osuus muodostaa ongelman staattisen analyysin pohjalta tietoturvaa arvioiville kehittäjille [15]. Nämä hälytykset eivät välttämättä ole tietoturvauhkia, mutta kehittäjän on silti arvioitava ne. Lisäksi SAST-järjestelmät eivät yleisesti tarjoa tarkkoja ratkaisuja ongelmiin, mikä johtuu järjestelmän rajoitteista ymmärtää koodin kontekstia ja sidonnaisuuksia. Tämä saattaa johtaa ehdotettujen ratkaisujen vaikeaan tulkittavuuteen. [15]

2.4.2 DAST-järjestelmä

Dynaaminen tietoturvatestauksenjärjestelmä (DAST) eroaa SAST-järjestelmästä niin, että siinä ei keskitytä lähdekoodin vaan toimivan järjestelmän tutkimiseen. DAST-järjestelmän tavoitteena on selvittää, miten ohjelma reagoi haitallisiin syötteisiin. Järjestelmä toimii erilaisten testien perusteella. Se ajaa testejä erilaisilla syötteillä

kohdejärjestelmään ja seuraa miten ohjelma reagoi syötteisiin. DAST testit simuloivat erilaisia tietoturvauhkia. DAST-järjestelmät hyödyntävät erilaisia skannaustyökaluja, rajapintakutsuja ja tunnettuja haitallisia koodipätkiä toteuttaessaan toimintaansa. Järjestelmät sisältävät yleensä valmiiksi asennettuja hyökkäysvektoreita ja antavat käyttäjälle mahdollisuuden määrittää omia hyökkäystestisettejä riippuen testattavan applikaation arkkitehtuurista. [16]

Ongelmana DAST-järjestelmissä on integraatio jatkuvaan kehitykseen. DAST-järjes-telmän toimintaan saaminen on hankalampaa verrattuna SAST-järjestelmään. DAST-järjestelmä tarvitsee yhteyden ohjelmaan ja sen päätepisteisiin, jotta se voi toimia. Yleisesti se voidaan implementoida ohjelmointihallintajärjestelmään, kuten GIT-järjestelmään, minkä jälkeen sitä voidaan ajaa koodille aina muutosten yhteydessä. DAST-järjestelmien toinen ongelma liittyy suoritustehoon. DAST-järjestelmiä halutaan lisätä automaation takia CI/CD-prosessiin. Järjestelmän lisääminen monimutkaistaa CI/CD-prosessia ja sen käyttöönottoa. Rangnau et al. mukaan DAST-järjestelmän lisääminen hidastaa tuotantoon viemistä. Heidän artikkelissaan kerrotaan, että suurin osa koodin tuotannollistamisen prosessissa menee DAST-järjestelmän käyttöönottoon. Kun järjestelmä on otettu käyttöön, se tuottaa nopeasti analyyseja. Testien tulokset ovat kuitenkin vähemmän yksityiskohtaisia verrattuna manuaalisen testauksen tuloksiin, ja ne pitää konfiguroida applikaatiokohtaisesti. [16]

2.4.3 IAST-järjestelmä

IAST-järjestelmä on kolmesta testausjärjestelmästä uusin ja vasta viime vuosina kehitetty tietoturvatestauksen järjestelmätyyppi. IAST-järjestelmien ideana on ottaa SAST- ja DAST-järjestelmistä niiden parhaat puolet ja toteuttaa ne CI/CD-prosessiin. Järjestelmä hyödyntää staattista koodianalyysia sekä dynaamisen testauksen testejä tutkiakseen tietoturvauhkia. Tämä on mahdollista IAST-järjestelmän käyttämän ainutlaatuisen asiayhteysmekanismin avulla. Mekanismi mahdollistaa staat-

tisen analyysin ja dynaamisen analyysin tiedon hyödyntämisen tuloksissa. Asiayhteysmekanismi on toteutettu näkökulmaohjatulla ohjelmointitekniikalla, joka mahdollistaa useamman datavirran käytön ja vertailun ohjelman sisällä. [13], [17]

Asianyhteysmekanisminsa ansioista IAST-järjestelmä erottuu DAST- ja SAST-järjestelmistä tarjoamalla heikkouksista tarkempaa analyysia. Setiawan et al. mukaan IAST on erityisen tehokas internetissä toimivien järjestelmien tutkimisessa, sillä se yhdistää sekä staattisen että dynaamisen analyysin edut. IAST:n etuna on, että se voidaan ottaa käyttöön ohjelmistokehityksen kaikissa vaiheissa: integroiduissa kehitysympäristöissä, jatkuvassa integraatiossa ja jopa tuotannossa. Haasteena on, että IAST vaatii pääsyn lähdekoodiin ja tuotantoon siirrettäessä ylimääräisen vaiheen dynaamisille testeille. IAST on verrattain uusi teknologia, ja sen käyttöön liittyvät järjestelmät ovat vielä harvassa. Tekniikan monimutkaisuus ja vaatimukset ovat myös korkeammat verrattuna muihin tutkitun asiakirjan järjestelmiin. [17]

2.5 RASP

RASP *Runtime application self protection* RASP, eli suorituksen aikainen turvaus, on tekniikka, jolla pyritään hallitsemaan sovelluksen ajonaikaista toimintaa tuotantoympäristössä. Toisin kuin kehitysvaiheessa käytettävät automaattiset turvajärjestelmät, RASP toimii sovelluksen ollessa aktiivisessa käytössä. Tekniikan päämääränä on havaita ja reagoida poikkeamiin tai ylimääräiseen kuormitukseen, jotka saattavat viitata turvallisuusongelmiin. RASP voi esimerkiksi tarkastella rajapintakutsuja ja analysoida niiden määrittelyjä, hallita tiedonhaun, tallentaa ja palauttaa tiedot oikein, sekä varmistaa prosessin sujuneen asianmukaisesti. [18]

RASP:n käytössä haasteena on nykyisten ohjelmien monimutkaisuus. Kun sovellukset vuorovaikuttavat muiden rajapintojen tai ympäristöjen kanssa, RASP ei välttämättä kykene hallitsemaan niitä kaikkia. Jos ohjelma esimerkiksi suorittaa vioittuneen objektin tai funktion, RASP ei välttämättä tunnista ongelmaa. Lisäksi

monia RASP:n tarjoamia turvatoimintoja voidaan saavuttaa käyttämällä muita tekniikoita, kuten sovelluspalomuureja tai resurssien monitorointia, jotka voivat suojata useita sovelluksia ilman ohjelmakohtaista määritystä. [18]

3 Ohjelmointirajapinnat

Tässä luvussa käsitellään ohjelmoitavia rajapintoja: niiden toimintaperiaatteita sekä turvallisuusmekanismeja. Lisäksi esitellään kymmenen merkittävintä tietoturvauhkaa, jotka toimivat perustana rajapintojen tietoturvan automatisoinnin tarkastelulle.

3.1 Ohjelmoitavan rajapinnan toiminta

Ohjelmoitavat rajapinnat ovat keskeisiä elementtejä ohjelmistojen, komponenttien tai moduulien välisessä vuorovaikutuksessa, sillä ne toimivat rajapintoina näiden eri osien välillä. Ohjelmoitavien rajapintojen perimmäinen tavoite on mahdollistaa sujuva kommunikaatio eri järjestelmien tai osien kesken. Tämä kommunikaatio ohjelmoitavien rajapintojen kautta voidaan luokitella kahteen päätyyppiin: informaation tarjoaviin sekä toiminnallisiin rajapintoihin. Informaation tarjoavat rajapinnat keskittyvät tiedon siirtoon osapuolten välillä, esimerkiksi mahdollistamalla järjestelmälle A tiedon hankkimisen järjestelmästä B rajapinnan määritelmän mukaisesti. Tämä prosessi voi sisältää tiedon selektiivistä jakamista, muokkausta tai prosessointia rajapinnan asetuksista riippuen. Toiminnalliset rajapinnat puolestaan tarjoavat konkreettisia toimintoja, jotka voidaan toteuttaa rajapinnan kautta. Ne mahdollistavat esimerkiksi sen, että ohjelmiston A voi käyttää karttasovelluksen kirjastoa ja muokata karttanäkymää hyödyntäen toiminnallisuuksia, joita rajapinta tarjoaa moduulilta B. Kommunikaation erottelu kahteen päätyyppiin korostaa ohjelmoitavien

rajapintojen monipuolisuutta ja niiden kykyä tukea erilaisia toiminnallisuuksia ja tiedonvaihtoa ohjelmistojen välillä. [1], [2].

Rajapintojen käyttö on levinnyt laajalle nykyaikaisessa ohjelmistokehityksessä niiden tarjoaman toiminnallisuuden ansiosta. Rajapinnat mahdollistavat rajatun mutta hallitun kommunikaation eri ohjelmistojen välillä. Kommunikaatio voi tapahtua sellaisten järjestelmien välillä, jotka sisältävät käyttäjälle salattua tietoa: yritykset voivat esimerkiksi vaihtaa asiakastietoja keskenään salatusti ja kontrolloidusti, siten että kumpikin osapuoli pääsee käsiksi vain toiminnan kannalta merkittäviin tietoihin toisen järjestelmässä. Internet-pohjaiset rajapinnat, jotka ovat kaikkien saavutettavissa internetin kautta, ovat erityisen suosittuja [19].

Rajapinnat luokitellaan vielä erikseen niiden näkyvyyden perusteella avoimiin, kumppani- ja sisäisiin rajapintoihin. Avoimessa rajapinnassa rajapinnan käyttäjällä ei ole mitään rajoitteita, ja yleensä nämä rajapinnat ovat tarjolla kaikille käyttäjille. Kumppanirajapinnoissa rajapinnat ovat suojattu erilaisilla lisensseillä tai salauksilla, joilla rajataan pääsy vain tämän rajapinnan käyttäjiin. Kumppanirajapinta mahdollistaa esimerkiksi arkaluontoisemman tiedon siirtämistä rajapinnan yli. Sisäinen rajapinta on nimensä mukaan sisäinen, ja siihen pääsy rajoittuu yleensä tietyn organisaation sisäiseksi. Sisäisessä rajapinnassa liikkuu yleensä yksityistä tietoa tai rajapinta mahdollistaa kriittisiä toimintoja. [20]

Ohjelmoitavien rajapintojen turvallisuuden tulee olla keskiössä riippumatta siitä, kuuluuko rajapinta avoimiin, kumppani- tai sisäisiin kategorioihin. Turvallisuuden perusta luodaan jo rajapinnan suunnitteluvaiheessa, jossa on olennaista huomioida rajapinnan käyttötarkoitus, käsiteltävä data tai toiminnot sekä kohdennettu käyttäjäryhmä. Tämän pohjalta on mahdollista tunnistaa potentiaaliset uhat ja kehittää strategioita niiden torjumiseksi. Haavoittuvuuksia voi aiheuttaa myös käytetty ohjelmointikieli sekä muut ohjelmointitekniikat. Lisäksi haavoittuvuustietokantojen hyödyntäminen voi auttaa tunnistamaan ja ehkäisemään mahdollisia uhkia. [21]

3.2 Ohjelmointirajapintojen uhat

Vuonna 2019 OWASP *(The Open Web Application Security Project)*tunnisti keskeiset uhat ohjelmoitaville rajapinnoille, jotka kattavat kirjautumiseen ja koodin suorittamiseen liittyvät turvallisuushaasteet. OWASP on yhteisöpohjainen hanke, joka julkaisee säännöllisesti päivityksiä verkkosovellusten suurimmista turvallisuusuhista, mukaan lukien rajapinnat, ja edistää yleistä tietoisuutta internetin turvallisuusasioista [22]. On tärkeää, että nämä uhat otetaan huomioon jo rajapinnan suunnitteluvaiheessa, sillä standardit protokollat ja niiden turvallisuustoimenpiteet eivät yksinään tarjoa riittävää suojaa [23]. Seuraavissa luvuissa käsitellään tarkemmin OWASP:n määrittelemiä uhkia ja niiden toimintamekanismeja.

3.2.1 Rikkinäisen objektitason valtuutus

Rikkinäisen objektitason valtuutus, englanniksi *Broken Object Level Authorization*, tunnistetaan yhtenä yleisimmistä internetin haavoittuvuuksista. Haavoittuvuudessa hyökkääjä saa pääsyn käyttäjille rajattuihin objekteihin rajapinnan tunnistautumisprosessin kautta. Tunnistautumisprosessi on toteutettu ohjelmoitavan rajapinnan lähdekoodissa tarkoituksena varmentaa käyttäjän tai objektin identiteetti. Identiteetin varmentamisen jälkeen rajapinta paljastaa käyttäjälle tai objektiryhmälle määritellyt ominaisuudet. Hyökkääjät löytävät tämän haavoittuvuuden tutkimalla sivuston lähdekoodia, rajapinnan yhteyttä, GET- ja POST-pyyntöjä tai rajapinnan osoitteita. [24]

Haavoittuvuudessa hyökkääjä pyrkii arvaamaan rajapinnassa käytettäviä avaimia käyttäjä- tai objektikohtaisesti. Tämä tapahtuu esimerkiksi hankkimalla pääsyn rajapintaan ja sitten muokkaamalla kutsun parametreja. Esimerkiksi ohjelmistolistauksessa 1 näemmä erään ohjelmoitavan rajapinnan kutsun. Tästä kutsusta voimme päätellä, että muokkaamalla JSESSIONID perässä olevaa kenttää, voimme kutsua rajapintaa eri käyttäjätiedoilla ja näin päästä käsiksi erilaisiin objektei-

hin. Kyseisessä esimerkissä vaihtamalla tunnisteen kayttaja1 tilalle kayttaja2 voidaan saada pääsyn käyttäjän 2 tietoihin. Hyökkäys on mahdollinen, sillä yleensä rajapinnan toteutuksen puolella käyttäjiä ja dataa käsitellään tunnisteiden kautta. Tunnisteet menevät yleensä sekaisin tai niitä voidaan käyttää suoraa sellaisenaan tietokannan arvoihin. Monimutkaisissa rajapinnoissa tunnisteet pysyvät samoina eri ohjelman komponenteissa, jolloin tätä haavoittuvuutta voi esiintyä ja sitä pystytään hyödyntämään rajapintaa vastaan. [24], [25]

Ohjelmalistaus 1 Rajapintakutsu, josta rikkinäinen objektitason valtuutus on helppo tunnistaa

```
GET http://localhost/testi/kauttaja.php?kauttaja=
Host: localhost/testi
User-Agent:Chrome/91.0.4472.124
Accept-Encoding:gzip
Referer: http://localhost/testi/
X-requested-With: XMLHttpRequest
Cookie: JSESSIONID=kayttaja1
Connection: close
```

Edellä kuvattua haavoittuvuutta vastaan on useampi tapa puolustautua. Yleisin on luoda epäsuora viittausjärjestelmä rajapinnan kutsujen ja itse datan välille. Käytännössä tämä tarkoittaa sitä, että luodaan tunnistekartta, jossa rajapinnasta saatu tunniste ei viittaa suoraan objektiin. Hyvänä tapana haavoittuvuutta vastaan on käyttää tarkempaa hallintaa eri käyttäjäoikeuksien välillä. Esimerkiksi kutsussa voidaan käyttäjää tunnistettaessa käyttää useampaa kenttää ja näin tunnistaa, mikäli kutsua on muokattu. Lisäksi käyttäjäoikeuksien rajoituksella voidaan varmistaa tietoturvallisuutta. Silloin käyttäjälle annetaan pääsy vain niihin tietoihin, mitkä ovat hänelle tärkeitä. Näin käyttäjä ei voi muokata tietoja, joihin hänellä ei ole oikeutta. Viimeiseksi kaikki järjestelmässä oleva data voidaan salata kryptografian avulla.

Tällöin avaimen tai tunnisteen paljastuessa hyökkääjä ei voi hyödyntää saamaansa dataa ilman salausavaimen purkua. [25]

3.2.2 Rikkinäinen käyttäjän tunnistaminen

Rikkinäinen käyttäjän tunnistaminen, englanniksi *Broken User Authentication*, on haavoittuvuus, joka johtuu puutteellisesta tunnistautumis- ja kirjautumisprosessien käsittelystä. Haavoittuvuudessa hyökkääjä voi hyödyntää useampaa hyökkäyskulmaa yksinkertaisesta salasanan arvaamisesta salasanalistan avulla tehtäviin teknisempiin hyökkäyksiin. Niihin kuuluvat esimerkiksi salausalgoritmin murtaminen tai käytössä olevien protokollien haavoittuvuuksien hyväksikäyttö. OWASP:n raportoimien tietojen mukaan, tällainen haavoittuvuus johtuu usein kirjautumisprosessien riittämättömistä suojausmekanismeista ja näiden mekanismien puutteellisista toteutuksista. Esimerkkejä riittämättömistä suojaustoimenpiteistä ovat rajapintojen päätepisteiden suojaamatta jättäminen tai pelkän rajapinta-avaimen käyttö suojausmenetelmänä. Ongelmat, jotka johtuvat puutteellisista toteutuksista, sisältävät muun muassa heikon salauksen käytön salasanoissa tai yksinkertaisten salasanojen sallimisen järjestelmässä. [22]

Rajapinnan suojaaminen tunnistautumisella ja vahvoilla mekanismeilla on tärkeää. Puutteellisesti suojattu rajapinta mahdollistaa hyökkääjille pääsyn muiden käyttäjien tilienä hallintaan. Tämä onnistuu esimerkiksi kutsumalla salasananvaihtorajapintaa toisen käyttäjän tiedoilla. Kun kutsu on tehty, hyökkääjä yrittää arvata väliaikaisen salasanan tai linkin, jonka avulla salasana vaihdetaan. Näin hyökkääjä saavuttaa pääsyn käyttäjätilille. Lisäksi haavoittuvuus voi antaa mahdollisuuden päästä käsiksi dataan. Tästä hyvän esimerkin antaa alkuvuodesta 2021 julkaistu artikkeli Parler-viestintäsovelluksesta, johon tehdyssä hyökkäyksessä päästiin käsiksi koko sovelluksen keräämään dataan. Tämä onnistui, koska hyökkääjät löysivät suojaamattomia rajapinnan päätepisteitä, jotka mahdollistivat dataan pääsyn. Sen seu-

rauksena hyökkääjät pystyivät lataamaan kaiken datan, mitä sovellus oli kerännyt eri käyttäjiltään. [26]

Suunnitteluvaiheessa täytyy ottaa huomioon, miten rajapinnan päätepisteet on suojattu protokollan ja tunnistautumisen tasoilla. Useiden tunnistautumismenetelmien käyttö, kuten salasanan ja käyttäjätunnuksen lisäksi esimerkiksi kertakäyttöisen salasanan tai dynaamisen päätepisteen käyttö, on suositeltavaa. Bisegna et al. mainitsevat teoksessaan finanssialan ohjelmoitavien rajapintojen automaattisen testauksen olevan avainasemassa tämän haavoittuvuuden estämiseksi. Automaattiset testit mahdollistavat hyökkääjien hyväksikäytölle mahdollisesti alttiiden heikkojen salasanojen ja suojaamattomien päätepisteiden havaitsemisen. [27]

3.2.3 Liiallinen tiedon altistuminen

Liiallinen tiedon altistuminen, englanniksi *Excessive Data Exposure*, on haavoittuvuus, joka mahdollistaa sensitiivisen datan pääsyn hyökkääjän käsiin rajapinnan kautta. Tämä haavoittuvuus ilmenee, kun sovelluksen ohjelmoitava rajapinta palauttaa käyttäjälle tarpeettoman määrän dataa. Näin tapahtuu järjestelmissä, joissa dataa prosessoidaan käyttöliittymän puolella. Käyttöliittymässä tapahtuvan datan prosessoinnin heikkoutena on, että hyökkääjä voi päästä käsiksi suoraan dataan analysoimalla sovelluksen ja rajapinnan välistä liikennettä. Tutkimalla liikennettä hyökkääjä voi löytää sensitiivistä tietoa, joka ei tavallisesti ole näkyvissä loppukäyttäjälle. [28] Hyökkääjän on helppoa tunnistaa haavoittuvuus tutkimalla ohjelmoitavan rajapinnan kutsujen vastauksia ja käyttöliittymän lähdekoodia sekä analysoimalla pakettiliikennettä. Ohjelmalistauksesta 2 havaitaan, miten tietokanta palauttaa liikaa tietoa. Kaikkea rajapinnalta tulevaa tietoa ei tarvitse näyttää loppukäyttäjälle. Ongelmana kuitenkin on, että kyseisestä rajapinnan palautuksesta hyökkääjän on helppo saada sensitiivistä dataa, kuten luottokortti- tai tilintiedot. [22]

Kuvattua haavoittuvuutta on hankala torjua automatisoiduilla testeillä, sillä tes-

Ohjelmalistaus 2 Luottokorttitietojen paljastuminen johtuen liiallisesta tiedosta rajapinnan vastauksessa

```
[
    {

        "Kauttajatunnus":"kauttaja1",
        "tilaustaso":"perustilaaja"
        "salasana":"Testi@123!"
        "luottotiedot":"1234567891011"
        "cvv":"1234"
        "korttistatus":"Voimassa"
    }
]
```

tit eivät osaa erottaa sensitiivistä dataa ja ohjelman tarvitsemaa dataa. Tärkein keino haavoittuvuuden ehkäisemiseksi on toteuttaa datan suodatus nimenomaan rajapinnan puolella eikä antaa tätä vastuuta kutsujalle, joka ei voi varmistaa datan eheyttä luotettavasti. On useita keinoja, joilla hyökkääjä pääsee käsiksi dataan. On keskeistä, että ohjelmoitavien rajapintojen suunnitteluvaiheessa määritellään tarkasti, mitä dataa kutsuja tarvitsee ja että kutsujalle palautetaan ainoastaan tämä data.

OWASP suosittelee käyttämään laajoissa järjestelmissä skeemapohjaista validointimekanismia rajapinnan vastauksissa, jotta sensitiivisen datan käsittelyä voidaan luokitella ja hallita [22]. Näin voidaan varmistaa, missä sensitiivistä dataa liikkuu, ja tarkistaa, käytetäänkö sitä kutsujan puolella ollenkaan. Ylläpitoa ja riskien tarkistusta tulisi tehdä, sillä rajapinnat kehittyvät jatkuvasti ja eteen voi tulla tilanteita, joissa sensitiivistä dataa alkaa vuotaa rajapintaan päivityksen yhteydessä.

3.2.4 Resurssien ja kuormituksenrajoittamisen puute

Englanniksi *Lack of Resources & Rate Limiting* on ohjelmoitavan rajapinnan haavoittuvuus, jossa hyökkääjä pyrkii aiheuttamaan palvelunestohyökkäyksen puutteellisen resurssien ja kutsuntamäärien rajoittamisen seurauksena. Hyökkääjä saa palvelunestohyökkäyksen aikaiseksi kuormittamalla verkkoa, prosessoria, muistia tai muita rajapinnan resursseja. Kun rajapinnalta loppuu resurssit, se ei enää pysty vastaamaan kutsuihin. Ohjelmalistauksesta 3 havaitaan, miten yksinkertaista kutsua muokkaamalla hyökkääjä saa rajapinnan antamaan hakutuloksia kymmenen kokoisen listan sijaan miljoonan kokoisena listana. Suuren listan kokoamiseen tarvittava resurssien kulutus voi ylikuormittaa palvelimen ja pysäyttää palvelun. [22]

Ohjelmalistaus 3 Alkuperäinen kutsu ja hyökkääjän muokkaama kutsu

```
//Tavallisen rajapinnankutsun POST-kutsun sisältö
{
"hakuparametri":"testi"
"max_palautus":"10"
"sivu_koko":"10"
}
//Hyökkääjän muokkaaman POST kutsun sisältö
{
"hakuparametri":"testi"
"max_palautus":"1000000"
"sivu_koko":"1000000"
}
```

Edellä kuvatun haavoittuvuuden estäminen onnistuu hyvällä suunnittelulla ja resurssien rajoituksella. Suunnitteluvaiheessa on tärkeää ottaa huomioon, mistä rajapintaa voidaan kutsua ja kuinka sitä voidaan rajoittaa. Yksi keino on sallia raja-

pinnan kutsuminen vain tietyltä IP-alueelta. Toinen keino on käyttäjäkohtaisten rajoitusten asettaminen tunnistautumisjärjestelmässä, jolloin yksittäinen käyttäjä voi kuormittaa rajapintaa vain ennalta määrättyyn rajaan asti. Rajapinnan arkkitehtuurin avulla on mahdollista optimoida resurssien käyttö. Esimerkiksi virtualisoinnin käyttö mahdollistaa virtuaalikoneelle asetettavien ominaisuuksien rajaamisen niin, ettei se voi kuluttaa kaikkia palvelimen resursseja. Lisäksi suuren kulutuksen aikana virtuaalikoneilla on mahdollista skaalata palvelun resursseja vastaamaan lisääntyneeseen kulutukseen. [22]

3.2.5 Rikkoutuneen funktiotason valtuutus

Rikkoutuneessa funktiotason valtuutuksessa *Broken function level authorization* hyökkääjä pyrkii hyödyntämään funktion puutteellisia tunnistautumismekanismeja. Tässä haavoittuvuudessa hyökkääjät käyttävät rajapinnan kutsuja, joissa käyttäjän tunnistus puuttuu kokonaan tai on toteutettu puutteellisesti: esimerkiksi henkilö ilman pääkäyttäjän oikeuksia voi arvata päätepisteen ja suorittaa kutsun, koska rajapinta ei tarkista kutsujan valtuuksia asianmukaisesti. Koodilistauksen 4 esimerkistä voimme tehdä kutsun rajapintaan ja poistaa sieltä käyttäjän. Hyökkääjä voi käyttää kyseistä rajapinnan päätepistettä hyväksi, sillä operaatiolle ei ole tehty valtuutusta. Eli kyseisestä sovelluksesta kuka tahansa voi poistaa käyttäjiä vain tietämällä poistofunktion päätepisteen. [22]

Tämän haavoittuvuuden torjuminen ei onnistu perinteisillä tietoturvamenetelmillä. Esimerkiksi, vaikka rajapintayhdyskäytävä voi tunnistaa kutsut, se ei kykene varmistamaan, onko kutsuja oikeutettu suorittamaan kyseisen toiminnon. Tekninen ratkaisu haavoittuvuuden torjumiseksi on suhteellisen yksinkertainen: päätepisteisiin tulisi lisätä tunnistekenttä, joka mahdollistaa kutsujan tunnistamisen. Tunnistuksen avulla voidaan varmistaa, että kutsujalla on tarvittavat oikeudet toiminnon suorittamiseen. On välttämätöntä, että eri päätepisteissä tarkistetaan, suorittaako

Ohjelmalistaus 4 Rajapintakutsu ei valtuuta kutsujaa mitenkään, mikä mahdollistaa kenen tahansa käyttäjän suorittaa kyseinen päätepiste

```
DELETE http://localhost/testi/kauttaja.php?kauttaja=kayttaja1
Host: localhost/testi
User-Agent:Chrome/91.0.4472.124
Accept-Encoding:gzip
Referer: http://localhost/testi/
X-requested-With: XMLHttpRequest
Connection: close
```

kutsun oikeutettu käyttäjä [28]

3.2.6 Massatoimeksianto

Massatoimeksiannossa *(Mass Assignment)* haavoittuvuus syntyy, kun käyttäjän syötteitä käsitellään harkitsemattomasti. Monissa ohjelmissa käytetään suoraan käyttäjältä saatuja syötteitä ilman niiden asianmukaista tarkistusta. Esimerkiksi työpaikan henkilötietolomakejärjestelmä käyttää suoraan käyttäjältä saatuja syötteitä, kuten puhelinnumeroa, kotiosoitetta tai palkanmaksutiliä, tietokannassaan. Tämän lisäksi sama rajapinta voi määrittää useampaa kenttää, jotka ovat piilossa loppukäyttäjältä. Tämä mahdollistaa hyökkääjälle pääsyn piilotettuihin kenttiin, kun hän lisää sopivia parametreja rajapinnan kutsuun. Seurauksena hyökkääjä voi muuttaa tietokannassa olevia tietoja, esimerkiksi saada pääkäyttäjäoikeudet. Tämän haavoittuvuuden hyväksikäyttö vaatii yleensä tarkempaa tietoa kohdejärjestelmästä. Se vaatii hyökkääjältä tietoa kohdejärjestelmästä ja sen rakenteesta. Hyökkääjä voi kokemuksen avulla arvata piilotettuja kenttiä. Esimerkiksi ohjelmistolistauksen 5 hyökkääjä keskeyttää PUT-kutsun ja lisää siihen muutaman kenttää ennen kutsun lähettämistä päätepisteelle ja pääse muuttamaan itsensä järjestelmän pääkäyttäjäksi. Tämä

on mahdollista, sillä hyökkääjällä oli tietoa rajapinnan rakenteesta. Lisäksi syötteen validointi puuttui päätepisteestä, mikä mahdollistaa haitallisen syötteen tekemät muutokset tietokantaan. [22]

Ohjelmalistaus 5 Järjestelmän odottama kutsu ja hyökkääjän muuntelema kutsu

```
{
"tunniste":"000123",
"rooli":"Tekninen asiantuntija",
"turvallisuustaso": "3",
"salasana":"--------",
}
//Hyökkääjän tekemät lisäykset kutsuun
{
"tunniste":"000123",
"rooli":"Tekninen asiantuntija",
"turvallisuustaso": "3",
"salasana":"--------",
"is_admin"="true",
"is_key_user"="true",
}
```

Massatoimeksiannon haavoittuvuuden tunnistaminen on tietoturvatasolla vaikeaa. Vaikka yksikkötestien avulla voidaan havaita ja tunnistaa haavoittuvuus, automatisoidut menetelmät eivät välttämättä tunnista sitä johtuen järjestelmien erilaisuudesta. Tietyt koodintarkistustyökalut voivat osittain havaita haavoittuvuuden, mutta eivät kykene tunnistamaan sitä täydellisesti. Samoista syistä haavoittuvuuden automatisoitu valvonta on vaikeaa toteuttaa. Haavoittuvuus voidaan tunnistaa suunnittelun ja toteutuksen aikana, mikäli vältetään suoraan käyttäjän syötteiden käyttämistä ohjelmassa. Käyttäjältä tulevat syötteet tulisi validoida huolellisesti en-

nen niiden käyttöä. Lisäksi ohjelman eri toiminnot voidaan suojata tunnistautumisen taakse. Esimerkiksi mainitussa tapauksessa on tärkeää varmistaa, että käyttäjällä on oikeasti oikeudet tehdä muutoksia järjestelmän pääkäyttäjäasetuksiin. Oikeuksien asianmukainen tarkistus mahdollistaa luvattoman toiminnan estämisen kohdejärjestelmässä. Uhan torjuminen vaatii järjestelmäsuunnittelua ja sen ottamista huomioon turvallisuussuunnitelmassa. [22]

3.2.7 Turvallisuuden väärä määritys

Turvallisuuden väärä määritys, englanniksi *Security Misconfiguration*, viittaa laajaan joukkoon haavoittuvuuksia, joita yhdistää virheet turvallisuuden määrityksissä tai liian yksinkertaisten turvallisuusmekanismien käyttö .Tällaiset virheet voivat mahdollistaa hyökkäykset. Esimerkkejä haavoittuvuuksista ovat puutteelliset *cross-origin*-turvallisuuspolitiikan määritykset, puutteelliset toteutukset tai turvattomien protokollien hyödyntäminen sovelluksessa. Tyypillisesti hyökkääjät hyödyntävät lisäturvatoimien, kuten *HTTPS* tai *TLS*, puuttumista. On tärkeää huomioida, että haavoittuvuudet voivat ilmetä missä tahansa, olipa kyse rajapinnan koodista tai eri *OSI*-mallin kerroksista. [22]

Haavoittuvuuksien havaitseminen ja tarkistaminen tulisi olla osana turvallisuuspolitiikkaa. Sovelluksessa käytössä olevia haavoittuvuuksia tulisi säännöllisesti arvioida tarkastelemalla konfiguraatiotiedostoja, rajapinnan komponentteja ja kolmansien osapuolien tarjoamia ratkaisuja. Automaation avulla pystytään löytämään yleisiä haavoittuvuuksia erilaisten listauksien avulla. Lisäksi eri protokolliin löytyy yleisiä ohjeita ja toimintatapoja, miten implementointi tehdään turvallisesti.

Turvallisuuden väärään määritykseen liittyvien haavoittuvuuksien havaitseminen ja tarkistaminen tulisi olla osana turvallisuuspolitiikkaa. Sovelluksessa käytössä olevia haavoittuvuuksia tulisi säännöllisesti arvioida tarkastelemalla konfiguraatiotiedostoja, rajapinnan komponentteja ja kolmansien osapuolien tarjoamia ratkai-

suja. Automaation avulla pystytään löytämään yleisiä haavoittuvuuksia erilaisten CVE *(Common Vulnerabilities and Exposures)* -listauksien avulla. Lisäksi eri protokolliin löytyy yleisiä ohjeita ja toimintatapoja siihen, miten implementointi tehdään turvallisesti.

3.2.8 Injektiohaavoittuvuus

Injection eli injektiohaavaittuvuus on yleinen haavoittuvuus, jota ilmenee kaikkialla internetissä pyörivissä sovelluksissa. Injektio hyökkäyksessä hyökkääjä pyrkii suorittamaan omaa koodia sovelluksessa tai saamaan sinne haitallista dataa. Haitallisella datalla hyökkääjä voi manipuloida sovelluksen tietokantaa. Tämä vaikuttaa sovelluksen toimintaan. Koodin injektiolla hyökkääjä mahdollistaa oman koodin suorittamisen sovelluksen sisällä, joka voi johtaa toisen haavoittuvuuden syntymiseen tai siihen että hyökkääjä saa tietoja sovelluksesta. Injektio voidaan piilottaa kutsun eri osiin, kuten evästeisiin, parametreihin tai tunnisteriveihin.

Injektiohaavoittuvuus (Injection) on yleinen haavoittuvuus, jota ilmenee kaikkialla internetissä pyörivissä sovelluksissa. Injektiohyökkäyksessä hyökkääjä pyrkii suorittamaan omaa koodia sovelluksessa tai lisäämään sinne haitallista dataa. Haitallisella datalla hyökkääjä voi manipuloida sovelluksen tietokantaa. Tämä vaikuttaa sovelluksen toimintaan. Koodin injektiolla hyökkääjä mahdollistaa oman koodin suorittamisen sovelluksen sisällä, mikä voi johtaa toisen haavoittuvuuden syntymiseen tai siihen, että hyökkääjä saa tietoja sovelluksesta. Injektio voidaan piilottaa kutsun eri osiin, kuten evästeisiin, parametreihin tai tunnisteriveihin. [22]

Ohjelmalistauksessa 6 hyökkääjä hyödyntää SQL-haavoittuvuutta ja lisää koodia rajapintakutsun sisään. Kun rajapinta suorittaa komennon kutsun sisäisellä parametrilla, se tulostaa kaikki rivit sovelluksen tietokannasta. Tämä tapahtuu koska OR-lause muuttaa mahdollisen rajapinnan kutsun "From account WHERE userid ..." niin, että ehtona oleva userid on aina totta. Tämä mahdollistaa sen, että hyök-

kääjä voi saada kaikki tiliin liittyvät tiedot. Muita yleisiä tapoja on esimerkiksi lisätä kutsuun exit() -lisäliite, joka yleensä lopettaa ohjelman tai kutsun suorituksen. [29]

Ohjelmalistaus 6 Hyökkääjä lähettää SQL-injektion sisältävän paketin rajapinnalle

```
Post http://localhost/testi/kauttaja
Host: localhost/testi
User-Agent:Chrome/91.0.4472.124
Accept-Encoding:gzip
Referer: http://localhost/testi/
X-requested-With: XMLHttpRequest
{
    userid: "12345’ OR 1=1 --"
}
```

Injektiohaavoittuvuuksien tunnistaminen sovelluksen kehitysvaiheessa on haastavaa. Haavoittuvuuden syynä voi olla muun muassa puutteellinen muuttujien käsittely tai heikot kirjastot. Staattiset koodianalyysityökalut voivat auttaa ehkäisemään joitakin näistä haavoittuvuuksista. Kaikkien mahdollisten haavoittuvuuksien testaaminen on kuitenkin vaikeaa, sillä nämä haavoittuvuudet ovat monimuotoisia ja niillä on kyky esiintyä kutsun missä tahansa osassa. Signatuuri- ja kuvioanalyysin avulla voidaan pyrkiä tunnistamaan ja estämään injektioita. Nämä menetelmät perustuvat normaalin kutsun tunnistamiseen ja sellaisten poikkeavien kutsujen erottamiseen, jotka eivät noudata tunnistettua kuviota tai jotka sisältävät poikkeavan signatuurin. OWASP suosittelee kaiken lähettäjältä tulevan datan validointia injektiohaavoittuvuuksien ehkäisemiseksi. [22]

3.2.9 Puutteellinen resurssien yläpito

Puutteellinen resurssien yläpito (*Improper assets management*) on suuri tietoturvan haavoittuvuuden aiheuttaja. Puutteellinen ylläpito tarkoittaa, että rajapintaan liittyvät dokumentaatiot ovat puutteellisia. Puutteellinen dokumentaatio voi jättää mainitsematta olemassa olevia parametreja tai päätepisteitä, jotka ovat käytössä mutta eivät näkyvissä dokumentaatiossa. Lisäksi ilman asianmukaista ylläpitoa esimerkiksi rajapinnan komponentteja ei ehkä päivitetä tai arvioida uudelleen parempien toteutustapojen löytämiseksi ajan myötä. Heikkoon ylläpitoon kuuluu myös puutteellinen lopetusstrategia rajapinnalle, mikä tarkoittaa, että suunnitelmaa rajapinnan poistamiseksi käytöstä ei ole tehty. Tämän seurauksena tietokannoissa voi pysyä avoinna päätepisteitä, jotka altistavat järjestelmän hyökkäyksille. [22]

Haavoittuvuuden estämiseksi olisi hyvä luoda standardoitu suunnitelma, joka kattaa rajapinnan elinkaaren eri pisteet etenkin huomioiden päivitysvälit ja lopetusstrategian. Suunnitelman tulisi pitää sisällään kuvaukset rajapinnasta sekä kaikista päätepisteistä, muista yhteyksistä ja parametreista, joita käytetään rajapinnassa. Automatisoituja dokumentointityökaluja on olemassa. Ne pystyvät luomaan tarvittavat dokumentaatiot. Rajapintoja tulisi seurata sovituin väliajoin ja tarkistaa erilaisilla haavoittuvuustyökaluilla. Staattisilla koodianalyysityökaluilla voidaan helposti etsiä tietoturvahaavoittuvuuksia rajapinnan eri komponenteista. Työkalut pystyvät kertomaan, mihin versioon kyseinen komponentti tulisi päivittää haavoittuvuuden ehkäisemiseksi. Rajapintojen aktiivinen monitorointi auttaa havaitsemaan epätavallista liikennettä ja tunnistamaan käytöstä poistettujen rajapintojen aktiivisena pysyneitä päätepisteitä, jotka voivat aiheuttaa tietoturvariskejä. [22]

3.2.10 Monitoroinnin ja lokituksen puute

Monitoroinnin ja lokituksen puute, englanniksi *Insufficient Logging & Monitoring*, on haavoittuvuus, joka mahdollistaa useamman eri haavoittuvuuden toteutumisen.

Tehokas monitorointi mahdollistaa potentiaalisten turvallisuusuhkien, kuten palvelunestohyökkäysten tai tietovuotojen, havaitsemisen rajapinnassa [22]. Monitorointi voi olla sekä sisäistä kohdistuen rajapinnan toimintaan ja sen liikenteeseen että ulkopuolista, jolloin etsitään julkisesta internetistä vuotaneita tietoja, kuten salausavaimia tai käyttäjätietoja. Lokituksen avulla voidaan dokumentoida ja analysoida rajapinnan tapahtumia, jolloin mahdolliset haavoittuvuudet tai epäilyttävät toimet voidaan tunnistaa. Esimerkiksi tallentamalla ja analysoimalla rajapintakutsuja voidaan selvittää, sisältävätkö ne epäilyttävää sisältöä. Lokitus mahdollistaa yksittäisten käyttäjien toimien tarkastelun, mikä voi paljastaa tietoturvaloukkauksia.

Haavoittuvuuden ehkäisemiseksi on välttämätöntä sisällyttää lokitus ja monitorointi järjestelmän suunnitteluun. On tärkeää määritellä, mitkä tapahtumat lokitetaan. Erityisen tärkeää on määritellä ne tapahtumat, jotka ovat kriittisiä järjestelmän turvallisuuden kannalta. Valitsemalla tärkeät tapahtumat lokitukseen voidaan helpommin tunnistaa mahdollinen väärinkäyttö. [22]

Lisäksi rajapinnalle tulisi suunnitteluvaiheessa harkita erilaisia monitorointijärjestelmiä, jotka vastaavat parhaiten sen tarpeita. Lokituksen ja monitoroinnin automatisointi voi merkittävästi parantaa turvallisuutta. Se mahdollistaa automaattiset hälytykset poikkeamista, mikä nopeuttaa reagointia hyökkäyksiin tai turvallisuusuhkiin.

4 Automatisoitujen järjestelmien kartoitus

Tässä luvussa käsitellään automatisoitujen tietoturvan testausohjelmien kirjoa, esitellään avoimen lähdekoodin projekteja ja tuodaan esiin alan kirjallisuudessa mainittuja kaupallisia ohjelmia. Ohjelmien valintaan vaikutti olennaisesti niiden soveltuvuus CI/CD-ympäristöön ja ilmaisversion olemassaolo. Kaikki tarkastellut ohjelmat on valikoitu alan kirjallisuudessa esiintyvien viittausten pohjalta.

4.1 Kuvaus automatisoidusta tietoturvaratkaisusta

On suunniteltu, että yritys integroi automaatiojärjestelmän CI/CD-prosessiin. Ensivaiheessa pyritään varmistamaan sen toimivuus rajapintojen kehitysympäristössä. Rajapintojen kehitysympäristön automatisointiin käytetään tällä hetkellä avoimen lähdekoodin Jenkins-järjestelmää, ja versionhallintaa hoitavat useat palveluntarjoajat sekä Atlassian-pohjainen Jira-järjestelmä. Automatisoitujen tietoturvaratkaisujen tavoitteena on tietoturvallisuusongelmien automaattinen tunnistaminen ja havaitseminen automaatioprosessin aikana. Tietoturvaratkaisujen on tarkoitus toimia edellä kuvatussa järjestelmäympäristössä ja tukea laajasti erilaisia tekniikoita. Järjestelmän on tunnistettava rajapintojen tietoturvauhkia, ja tulosten tulee olla kehittäjille selkeästi tulkittavissa.

4.2 Penetraatiotestaus DAST-järjestelmillä

Tässä osiossa käsitellään DAST-järjestelmiin kuuluvia sovelluksia, jotka on valittu testauskäyttöön. Kaikkia näitä sovelluksia yhdistää niiden käyttö penetraatiotestauksessa.

4.2.1 OWASP ZAP

OWASP ZAP on OWASP Foundationin kehittämä DAST-pohjainen testausjärjetelmä. Kyseessä on vapaan lähdekoodin ohjelmisto, joka on maksuton kaikille käyttäjille. Järjestelmä on tarjolla sekä itsenäisenä sovelluksena että rajapintapohjaisena työkaluna, jonka voi integroida muihin järjestelmiin, esimerkiksi Jenkinsin-automaatioympäristöön. OWASP ZAP on ominaisuuksiltaan monipuolinen: se tarjoaa laajan valikoiman konfiguraatioita ja mahdollisuuden luoda omia testejä. OWASP ZAP toimii automaatiojärjestelmässä rajapintana, joka on käytettävissä myös erikseen asennettavana versiona. Rajapintatoiminnot ovat keskeisiä automatisoidun käytön kannalta. Ohjelman asentaminen on yksinkertaista, ja ZAP:n verkkosivuilta löytyy kattavasti ohjeita sen käyttöön. [5]

ZAP toteuttaa erilaisia hyökkäystyyppejä, jotka voidaan luokitella penetraatiotestauksen menetelmiin. Dokumentaation mukaan se kykenee suorittamaan esimerkiksi injektiohyökkäyksiä ja hyökkäyksiä, jotka kohdistuvat sovelluksen objekti- tai funktiotasoon. Saadut ilmoitukset hyökkäyksistä luokitellaan viiteen eri kategoriaan: korkea, keskitaso, alhainen, informatiivinen ja väärä hälytys. Hyökkäyksiä, jotka eivät aiheuta reaktiota sovelluksessa, ei luokitella. Korkean, keskitason ja alhaisen riskin kategoriat perustuvat OWASP:n haavoittuvuusraporttien arviointikriteereihin. Informatiiviset ja väärät hälytykset tarjoavat tietoa potentiaalisista sovellusongelmista, jotka voivat johtaa muihin ongelmiin. Järjestelmä ei esitä korjaustoimenpiteitä ongelmiin, vaan antaa yleiskuvauksen, josta voidaan johdattaa korjausehdotuksia muita työkaluja käyttäen. [5], [30]

4.2.2 Arachni Scanner

Arachni Scanner on Ruby-kielinen avoimen lähdekoodin DAST-järjestelmä, joka on suunnattu internetsovellusten haavoittuvuuksien tunnistamiseen. Tämän avoimen lähdekoodin projektin etuna pidetään se luomaa mahdollisuutta laajan yhteisön osallistumiselle lähdekoodin tarkasteluun ja kehittämiseen. Useat eri yritykset käyttävät Arachnia, mikä osoittaa sen soveltuvuuden erilaisiin ympäristöihin. Arachni tukee monipuolisesti eri käyttöjärjestelmiä ja alustoja mahdollistaen käytön komentoriviltä, web-pohjaisena sovelluksena tai osana CI/CD-prosessia. [5], [31]

Arachni kykenee tunnistamaan haavoittuvuudet, jotka on listattu OWASP:n uhkaluettelossa sekä muut yleiset internetsovellusten haavoittuvuudet. Se luokittelee havaitut uhat neljään kategoriaan, joista kolme ensimmäistä kategoriaa edustaa eri uhkatasoja ja neljäs keskittyy ohjelman toimintaan liittyviin huomioihin. Arachni antaa myös CVE-kuvauksen tunnistetuista haavoittuvuuksista, mikä helpottaa uhkien arviointia ja korjaamista. [31]

4.2.3 Wapiti

Wapiti on avoimen lähdekoodin DAST-järjestelmä. Se on suunniteltu internetsovellusten haavoittuvuuksien testaamiseen. Wapiti hyödyntää fuzz-testaustekniikkaa. Tämä menetelmä skannaa sovelluksen eri päätepisteet ja syöttää niihin satunnaista dataa tarkoituksena havaita järjestelmän reaktiot siihen. Tekniikka pyrkii jäljittelemään potentiaalisen hyökkääjän toimintaa. Wapiti tunnistaa yleisimmät sovellushaavoittuvuudet ja tarjoaa näin kattavan työkalun tietoturvatutkimukseen. [32]

Wapitin komentorivikyvykkyydet ovat edistyneimmät verrattaessa niitä muihin vastaaviin järjestelmiin, kuten ZAP-järjestelmään ja Arachniin. Testi suoritetaan määrittelemällä kohdesovelluksen aloituspiste, minkä jälkeen Wapiti etsii kaikki portit ja aliosoitteet. Tämän jälkeen Wapiti suorittaa testit järjestelmällisesti jokaiseen kohteeseen. Kuvassa 4.1 esitetään esimerkki Wapitin generoimasta tuloksesta, joka

Kuva 4.1: Wapiti työkalun visualisointi löydetystä haavoittuvuudesta

sisältää kommenttirivikomennon ja HTML-pyynnön, jonka avulla haavoittuvuus on todettu. Tämä antaa loppukäyttäjälle mahdollisuuden arvioida havaittua haavoittuvuutta itsenäisesti. [32]

4.2.4 Kaupalliset DAST-järjestelmät

Tässä osiossa aiemmin mainittujen avoimen lähdekoodin projektien lisäksi huomiota kiinnitetään kaupallisiin DAST-järjestelmiin, jotka ovat saatavilla erilaisten lisenssien kautta. Monet kaupalliset järjestelmät tarjoavat samoja ominaisuuksia kuin avoimenlähdekoodin vastaavat. Erona avoimen lähdekoodin järjestelmiin on usein se, että kaupallinen toimija tarjoaa ohjelmiston lisäksi apua käyttöönottoon ja tukea ongelmatilanteisiin. Alan kirjallisuuden perusteella suosittuja kaupallisia järjestelmiä ovat Netsparker ja Acunetix.

Netsparkerin ja Acunetixin omien ilmoitusten mukaan molemmat järjestelmät kykenevät tunnistamaan kaikki merkittävimmät tietoturvahaavoittuvuudet. Tarjolla on useita käyttövaihtoehtoja, mukaan lukien rajapintapohjaiset ratkaisut ja komentorivityökalut. Käyttöliittymien on kerrottu olevan yksinkertaisia ja helppokäyttöisiä. Yritysten markkinointimateriaalin perusteella molemmat järjestelmät ovat saaneet laajasti suuria yrityksiä asiakkaikseen. Kummastakaan järjestelmästä ei kuitenkaan ole saatavilla puolueetonta julkaisua, joten niiden ominaisuuksien vertaileva tarkastelu luotettavasti ei ole mahdollista. Saatavilla on vain yritysten itsensä tuottamaa aineistoa. [33], [34]

4.3 Lähdekoodin testaus SAST-järjestelmillä

Tässä kappaleessa esitellään SAST-järjestelmiä vertailua varten. Valintaperusteet ovat samat kuin DAST-järjestelmien kohdalla.

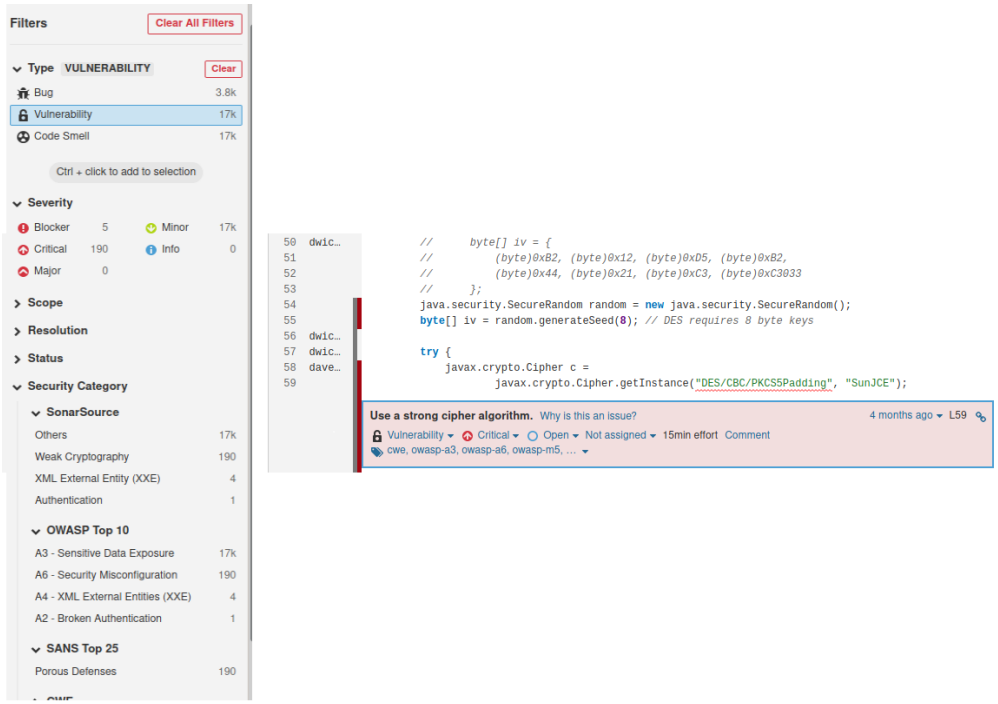
4.3.1 SonarQube

SonarQube on kaupallinen SAST-pohjainen työkalu. Työkalusta on saatavilla ilmaisversio, ja sitä käytetään tässä vertailussa. SonarQube toimii palvelinpohjaisena. Se suorittaa lähdekoodin analyysin koodin kokoamisvaiheen jälkeen. SonarQube tarjoaa tuen useille ohjelmointikielille ja kirjastojen tietoturvatarkistuksen hyödyntäen haavoittuvuustietokantoja. Analyysin jälkeen työkalu tuottaa interaktiivisen raportin, joka luokittelee löydökset bugeiksi, haavoittuvuuksiksi ja huonoksi koodiksi tarjoten samalla korjausehdotuksia. SonarQube sisältää myös kehittäjän IDE-järjestelmään integroitavia työkaluja, jotka tarjoavat lisäominaisuuksia haavoittuvuuksien torjuntaan koodin kehitysvaiheessa kehittäjän omassa ympäristössä. Näitä työkaluja ei kuitenkaan tässä työssä käsitellä tarkemmin. [35]

SonarQube mahdollistaa koodin luokittelun monin eri tavoin, mukaan lukien uhkaavuusluokitukset ja hälytykset koodin ongelmakohdista. Erilaisten luokitusten suurta määrää loppukäyttäjä voi suodattaa loppuraportista. Sovellus pystyy tunnistamaan ja linkittämään uhkat CVE-tietokantaan, mikä helpottaa haavoittuvuuksien ymmärtämistä ja korjaamista. Kuvasta 4.2 nähdään, että työkalu nostaa esille tarkat kohdat koodista, joissa uhat tai virheet havaitaan.

4.3.2 FluidAttacks Scan SAST-versio

FluidAttacks-tietoturvayhtiö tarjoaa työkaluja tietoturvan varmistamiseen eri alustoilla suosien vapaan lähdekoodin lisenssiä. Heidän päätuotteensa, FluidAttack, palvelee sekä DAST- että SAST-järjestelmissä. Työkalu ei kuitenkaan ole IAST-järjestelmä,



Kuva 4.2: SonarQube-järjestelmän visualisointi, joka esittää erilaisia luokituksia tulosten tulkintaa varten.

koska se ei toistaiseksi kykene yhdistämään SAST- ja DAST-analyysejä. Tässä vertailussa keskitytään SAST-versioon. Ohjelma tukee useita ohjelmointikieliä ja hyödyntää tunnettuja uhkatietokantoja. CI/CD-integraatio on tuettu joko valmiina virtuaalipalvelinratkaisuna tai mukautettuna asennuksena. Mukautetun asennuksen voi asentaa esimerkiksi omaan palvelinympäristöön. Palvelinympäristössä tarvitaan tuki kolmannen osapuolen NIX-ohjelmistoalustaan. [36]

Kuva 4.3: FluidAttack-SAST-työkalun tarjoama visualisointi haavoittuvuudesta

Järjestelmä antaa tulokset CSV-muodossa loppukäyttäjälle. Kuvasta 4.3 huomataan, että järjestelmä ei anna CVE-luokitusta tai muuta yleistä viitettä ongelmasta. Työ-kalu kuitenkin tarjoaa ongelmalle yleisen nimen ja sijainnin koodissa. Tämä esitystapa voi vaikeuttaa tulosten kategorisointia ja analysointia verrattuna muihin järjestelmiin. Toisaalta se mahdollistaa mukautetun näkymän luomisen käyttäjälle tai integraation kolmannen osa-puolen työkaluihin.

4.3.3 Nodejsscan

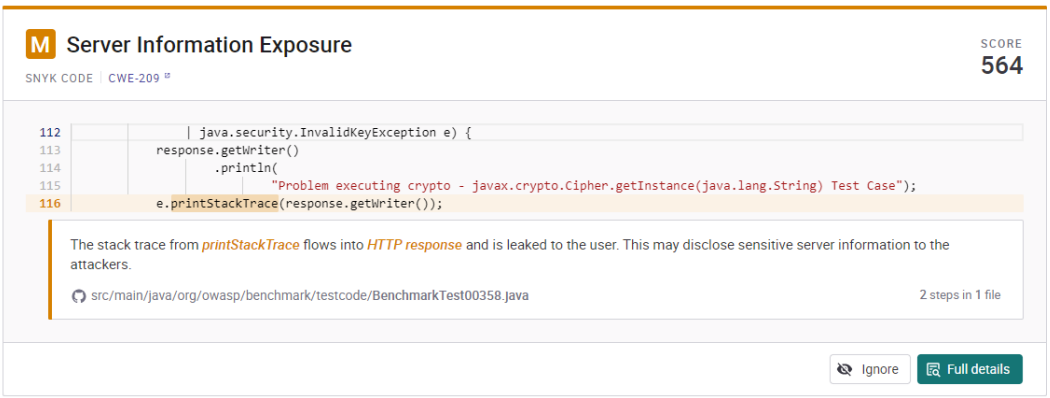
Nodejsscan on avoimen lähdekoodin projekti, joka keskittyy Node-tekniikkaan pohjautuvien sovellusten SAST-analyysiin. Node-tekniikka perustuu JavaScript -ohjelmointikieleen, joka on erittäin suosittu ohjelmointikieli. Työkalun kehittäjä on tietoturvatutkija Ajin Abraham [37]. Työkalu erottuu suurten yhtiöiden ja organisaatioiden tuottamista järjestelmistä siinä, että se tarjoaa monipuolisia käyttöönottoon liittyviä vaihtoehtoja.Nodejsscan voidaan asentaa virtuaalipalvelimelle, tai sitä voidaan käyttää komentoriviltä. Työkalu on maksuton, ja sen lähdekoodi on muokattavissa käyttäjän tarpeisiin.

Nodejsscanin rajoitus liittyy sen kykyyn analysoida ainoastaan JavaScript -projekteja. Tämän vuoksi sen pääasiallinen käyttökohde on Node-tekniikka, joka perustuu JavaScriptiin. Se ei kykene tunnistamaan muiden ohjelmointitekniikoiden haavoittuvuuksia. Siitä huolimatta Nodejsscan luokittelee haavoittuvuudet OWASP -standardeja noudattaen, toisin kuin muut tässä työssä esitellyt SAST-järjestelmät. Työkalu sisältää logiikan, joka ehdottaa korjauksia perustuen CVE-dokumentaatioon. [37]

4.3.4 Snyk

Snyk on viimeinen tässä vertailussa esitelty SAST-tekniikan järjestelmä. Sen toiminnallisuudet ovat samankaltaisia SonarQube-järjestelmän kanssa: molemmat tarjoa-

vat lisätoimintoja SAST-analyysin lisäksi. On huomionarvoista, että lisätoimintojen käyttö saattaa vaatia lisämaksua. Snykin tarjoama alusta integroituu suoraan versionhallintajärjestelmiin tukien CI/CD-prosessia. Snyk ei tarjoa mahdollisuutta ajaa järjestelmää käyttäjän omilla palvelimilla. Snyk luokittelee haavoittuvuudet vakavuuden mukaan kahteen tasoon ja käyttää sisäistä pisteytysjärjestelmää vakavuuden määrittämiseen. Työkalu mahdollistaa ongelmien seurannan versionhallinnassa, ja se pystyy luomaan valmiita korjausehdotuksia projektiin, jotka kehittäjän täytyy vain hyväksyä. Tämä voi kuitenkin kehittäjän näkökulmasta tuoda haasteita, jos kehittäjä ei itse ymmärrä luotua koodia. [38]



Kuva 4.4: Snyk-järjestelmän visualisointi havaitusta haavoittuvuudesta

Kuvassa 4.4 esitetään Snyk-järjestelmän tuottama haavoittuvuuden luokittelu, josta ilmenevät haavoit-tuvuuden sijainti ja potentiaaliset seuraukset. Järjestelmä antaa CVE-luokituksen ja Full Details -näkymässä korjausehdotuksia, jotka perustuvat avoimen lähdekoodin projektei-hin. Ehdotetut ratkaisut eivät ole sidonnaisia projektin kontekstiin, mikä tarkoittaa sitä, että ne eivät suoranaisesti ratkaise ongelmaa. Ehdotetut korjaukset voivat johtaa erillisen haaran luomiseen projektiin, tai ne voivat luoda bugitikettejä tuettuihin projektinhallintajärjestelmiin. Tämä ominaisuus on kuitenkin saatavilla vain kaupallisessa versiossa. [38]

4.3.5 Kaupalliset SAST-järjestelmät

Monia SAST-järjestelmiä tarjotaan kaupallisina versioina. Myös tutkielmassa aiemmin esitellyt ilmaisjärjestelmät SonarQube, FluidAttack ja Snyk sisältävät kaupalliset versiot, jotka tarjoavat laajempaa tukea käyttöönotossa ja ylläpidossa. Tässä osiossa verrataan GitLabin ja Kiuwanin kaupallisia SAST-sovelluksia.

GitLab tarjoaa versionhallintajärjestelmään integroituvan SAST-järjestelmän, joka liittyy tiiviisti kehitysprosessiin. Työkalun on havaittu olevan toiminnallisuuksiltaan samankaltainen Snykin kanssa, mukaan lukien korjausehdotusten käsittely sekä integraatio versionhallintajärjestelmään. GitLab edellyttää kuitenkin, että käytössä on sen tarjoama versionhallintajärjestelmä.

Kiuwan, toinen kaupallinen SAST-järjestelmä, on saanut kattavasti huomiota demonstraatioissaan. OWASP Benchmark -testissä Kiuwan saavutti täydellisen tuloksen löytäen kaikki haavoittuvuudet ilman vääriä hälytyksiä. Erikoiseksi Kiuwanin tekee sisäänrakennettu SCA (Static Code Analysis) -ominaisuus, joka toimii yhdessä SAST-toimintojen kanssa. Kiuwanin SCA-ominaisuus on useiden lähteiden mukaan arvioitu alan parhaimmaksi. [39]

4.4 IAST-järjestelmät

IAST-järjestelmiä on tullut markkinoille viime vuosina. Netsparker, Acunetix, Hdiv Detection ja Seeker ovat merkittävimpiä IAST-sovelluksia, jotka ovat saatavilla ainoastaan kaupallisilla lisensseillä. Tällä hetkellä ilmaisia versioita ei ole saatavilla testattavaksi. [4], [40]

Vaikka puolueettomia vertailuja näistä tuotteista ei ole saatavilla, yritykset ovat julkaisseet omia demojaan. Demojen perusteella nämä järjestelmät ovat osoittautuneet tehokkaiksi uhkien tunnistamisessa ja kattavien raporttien luomisessa. On kuitenkin huomattava, että ilmaiset demot ja videot käyttävät yksinkertaistettuja

projekteja, jotka on suunniteltu mainostarkoituksiin. [4], [40]

Puuttuvista vertailuista huolimatta voidaan päätellä, että IAST-tekniikka on tehokasta yhdistäessään SAST- ja DAST-järjestelmien toiminnallisuudet. Tekniikka on kuitenkin suhteellisen uutta ja kehityksen alla. Avoimen lähdekoodin ratkaisujen kehitys tulevaisuudessa on erityisen mielenkiintoista.

5 Tutkimuksen toteuttaminen

Tässä luvussa suoritetaan käytännössä viitekehyksessä esiteltyjen automatisoitujen tietoturvan testausjärjestelmien testaus. Järjestelmien testausprojekti ja testausympäristö tuodaan esille.

5.1 Tutkimustehtävä ja kysymykset

Tämän tutkimuksen tehtävänä on tarkastella kahta pääkohdetta. Ensinnäkin selvitetään, millaisia automaattisen tietoturvatestauksen järjestelmiä on saatavilla, kun resurssit ovat rajalliset. Toiseksi arvioidaan erilaisten järjestelmien kykyä tunnistaa haavoittuvuuksia rajapintaohjelmistossa.

Tietoturvatestauksen vaativuus johtuu siihen liittyvästä laajaan osaamiseen ja merkittävien resursien tarpeesta. Sovellusten käyttäjät odottavat niiden tietoturvan olevan virheetön. Projekteissa eteen tulee valinta uusien ominaisuuksien ja tietoturvan välillä; uusien ominaisuuksien kehittäminen on usein yritykselle tuottavampaa. Tässä tutkielmassa pyritään esittämään menetelmiä, joilla projektin tietoturvatason voi saavuttaa resurssitehokkaasti ketterässä kehitysympäristössä. Tästä lähtökohdasta kaksi tutkimuskysymystä muodostetaan.

Tutkimuskysymykset ovat:

1. Millaisia avoimeen lähdekoodiin perustuvia automaattisen tietoturvatestauksen järjestelmiä on saatavilla?

(a) Miten nämä järjestelmät soveltuvat CI/CD-ohjelmistokehitykseen?

2. Miten avoimeen lähdekoodiin perustuvat automaattisen tietoturvatestauksen järjestelmät soveltuvat rajapintojen haavoittuvuuksien tunnistamiseen?

5.2 Tutkimusmetodi

Tutkimuksen alkuvaiheessa oli välttämätöntä määrittää, mitä tietoturvatestausohjelmistoja on saatavilla. Tarvittiin käsitys siitä, miten eri ohjelmat toimisivat ja millaisia syötteitä ne tarjoaisivat. Havaittiin, että tietoturvatestauksen ohjelmistoille ei ole olemassa erityistä standardia. Ohjelmistotoimittajat ovat kehittäneet esimerkkejä, joiden avulla omaa ohjelmistoa esitellään. Tämän seurauksena vertailukelpoisten tulosten saavuttaminen muodostui vaikeaksi, sillä kukaan ei ole kehittänyt yhteistä standardia testiä, jota kaikki käyttäisivät. Valintaprosessi kohdistui lopulta OWASP Benchmark- ja Wavesep-järjestelmiin [41], [42]. Valinta osui lopulta OWASP Benchmarkiin, sillä sen lähdekoodi osoittautui ajantasaisemmaksi ja paremmin ylläpidetyksi verrattuna Wavesep-järjestelmään.

Testaamiseen käytettiin OWASP tuottamaa *OWASP Benchmark* järjestelmää [42]. Se on kehitetty standardisoimaan automatisoitujen tietoturvajärjestelmien testausta ja toimimaan testinä, jonka avulla voidaan verrata ohjelmia keskenään. Benchmark on internetapplikaatio, johon on toteutettu erilaisia haavoituvuuksia. Nämä haavoittuvuudet on valittu eri CWE-haavoittuvuusluokista. Tarkempi sisältö haavoittuvuuksista löytyy taulukosta 5.1.

Benchmark-ohjelma on kehitetty ensisijaisesti Java-ohjelmointikielellä, mutta se sisältää elementtejä myös HTML- ja JavaScript-ohjelmointikielistä. Työkalun ylläpidosta Githubissa vastaa OWASP -organisaatio. Kyseessä on avoimen lähdekoodin projekti, mikä mahdollistaa sen, että kuka tahansa voi käyttää ja muokata työkalua omiin tarpeisiinsa. Työkaluun on integroitu yleisimpiä uhkia CWE (Common Weak-

Haavoittuvuus	CWE
Komentoinjektio	78
Heikko kryptografia	327
Heikko hajautusalgoritmi	328
LDAP-injektio	90
Hakemiston virheellinen rajoitus	22
Evästeiden suojaus	614
SQL-injektio	89
Luottamusrajan rikkominen	501
Heikon satunnaisuuden käyttö	330
XPATH-injektio	643
XSS	79

Taulukko 5.1: Benchmark-ohjelman haavoittuvuuskuvaukset ja CWE-luokat [42]

ness Enumeration) -tietokannasta, mukaan lukien luvussa 3 käsitellyt uhat. Työkalu tarjoaa myös tiedot kaikista sen sisältämistä haavoittuvuuksista, mikä mahdollistaa tulosten arvioinnin. [42]

Benchmark-ohjelman keskeinen konsepti on, että se sisältää laajan valikoiman haavoittuvuuksia ja työkaluja, jotka mahdollistavat erilaisten raporttien analysoinnin ja vertailun. Sen tarkoitus on, että testattu automaatiojärjestelmä tuottaa haavoittuvuusraportin, jota voidaan verrata Benchmarkin tarjoamaan raporttiin. Tämän vertailun avulla voidaan arvioida automaatiojärjestelmän tehokkuutta haavoittuvuuksien tunnistamisessa. Näin ollen voidaan mitata, onko automaatiojärjestelmä tunnistanut haavoittuvuudet oikein, merkinnyt haavoittuvuuksia väärin tai luokitellut ne virheellisesti. [5], [42]

5.3 Tutkimuksen testausympäristö

Tutkimuksen testausympäristö esitellään tässä luvussa. Valintana oli virtuaalikoneratkaisu, joka pyrittiin saamaan vastaamaan Continuous Integration/Continuous Deployment (CI/CD) -putkien käyttämää virtualisaatiota. Tämä tarkoittaa, että ympäristö koostui useasta virtuaalikoneesta. On merkittävää, että DAST ja SAST -järjestelmien testaaminen vaatii erityiset järjestelyt. Luotu testausympäristö mahdollistaa useiden DAST- ja SAST-järjestelmien tehokkaan vertailun yksittäisessä projektissa. Ohjelmistotuotannon näkökulmasta tämä ympäristö ei kuitenkaan olisi kovin skaalautuva.

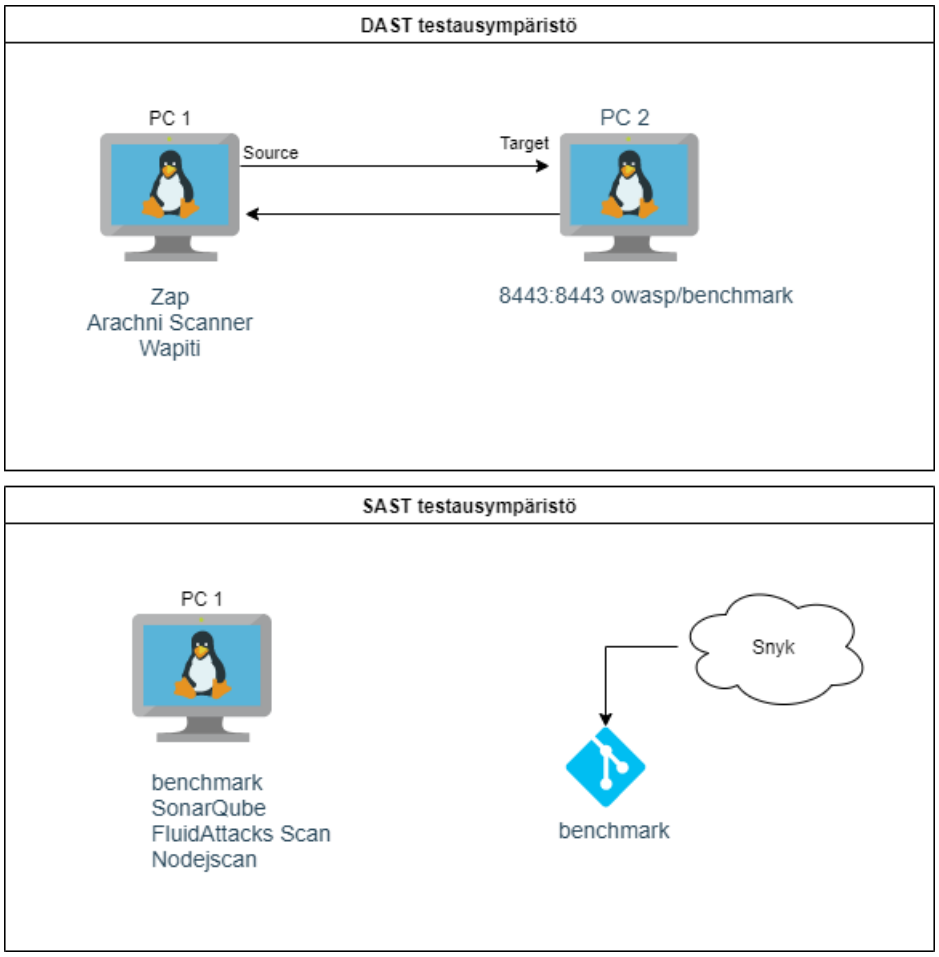
Virtuaalikoneiden käyttöjärjestelmiksi valittiin Linux-pohjaiset käyttöjärjestelmät Ubuntu 20.04 LTS PC 1:lle ja RHEL 8 PC 2:lle. Ubuntu valittiin PC 1:lle, koska se tuki kaikkia DAST-järjestelmiä, kun taas RHEL 8 valittiin PC 2:lle, sillä se on yleinen käyttöjärjestelmä internetpohjaisille tuotantopalvelimille. Virtuaalikoneet määritettiin samaan verkkoon, jotta ne voivat löytää toisensa.

DAST-järjestelmien testausta varten asennettiin kaksi virtuaalikonetta. PC 2:lle asennettiin Benchmark-ohjelma porttiin 8443, joka oli avattu verkkoon. PC 1:lle asennettiin DAST-järjestelmät ZAP, Arachni Scanner ja Wapiti, jotka sitten testasivat Benchmark-ohjelmaa PC 2:ssa avoimessa portissa suorittaen HTTP-kutsuja eri päätepisteisiin. Ohjelmien suoritettua testauksensa tulokset tallennettiin.

SAST-järjestelmien testaus toteutettiin PC 1:llä. SAST-järjestelmille riitti pääsy tiedostojärjestelmässä olevaan koodiin, minkä vuoksi Benchmark-ohjelman lähdekoodi asennettiin tähän laitteeseen. PC 1:lle asennettiin myös SonarQube, FluidAttacks Scan ja Nodejsscan, jotka ohjattiin analysoimaan projektin tiedostoja tiedostojärjestelmässä.

Poikkeuksena Snyk-järjestelmä ei analysoi koodia suoraan tiedostojärjestelmästä, vaan se toimii Snykin palvelimella ja analysoi koodin versionhallintajärjestelmästä. Testiä varten Benchmark-projekti siirrettiin GitHub-kirjastoon ja Snyk ohjattiin

analysoimaan sitä.



Kuva 5.1: SAST ja DAST-järjestelmien testaamiseen käytetyt ympäristöt.

Kuvassa 5.1 on esitetty SAST- ja DAST-järjestelmien testausympäristöt. Kuvassa havainnollistetaan myös eroja DAST- ja SAST-testausjärjestelmien välillä. DAST-järjestelmien suorittamat tutkivat hyökkäykset on esitetty kuvassa nuolena PC 1:stä PC 2:een. PC 2:sta palautuva nuoli kuvaa palautuvaa HTTP-kutsua, jonka perusteella järjestelmä arvioi mahdollisen haavoittuvuuden olemassaolon. SAST-järjestelmien testaus vaati vain yhden virtuaalikoneen SonarQuben, FluidAttacks Scannin ja Nodejsscanin osalta, kun taas Snyk analysoi Benchmark-järjestelmän lähdekoodia GitHubissa sijaitsevasta versiosta.

6 Tulokset

Tässä luvussa esitellään tuloksia, joita kerättiin testiajoissa Benchmark-ohjelmaa vastaan. Tulokset käsitellään pääluvuittaan SAST-, DAST- ja IAST-ohjelmien välillä.

6.1 Tulosten keräys

Tulostenkeräämismenetelmänä käytettiin manuaalista lähestymistapaa. Tämä johtui siitä, että Benchmark-projekti sisälsi useita tutkimuksen ulkopuolelle jääviä haavoittuvuuksia, mikä esti automaattisen raportoinnin hyödyntämisen. Manuaalisen strategian valintaan vaikutti myös se, että kaikki vertailussa mukana olevat ohjelmat eivät tukeneet Benchmarkin raportointityökalua. Benchmarkin raportointityökalu on nimittäin suunniteltu tukemaan vain CSV- ja XML-muotoisia haavoittuvuusraportteja, joita ei voida tuottaa kaikista järjestelmistä.

Tulosten vertailua varten oli tarpeen kehittää pisteytysjärjestelmä. Pisteytysasteikko ulottui yhdestä sataan, ja täydet pisteet saavutettiin, kun ohjelma tunnisti kaikki Benchmark-järjestelmän haavoittuvuudet ilman virheellisiä positiivisia tai negatiivisia tuloksia. Arvioinnissa huomioitiin myös työkalun käyttöönoton helppous CI/CD-putkessa sekä haavoittuvuuksien esittämisen selkeys loppukäyttäjälle

Tietyt kategoriat jäivät vertailun ulkopuolelle, jos ne eivät sisältyneet kolmannessa luvussa esitettyihin haavoittuvuuksiin. Injektiohyökkäykset ja valtuutuksen hallinnan puutteet yhdistettiin yhdeksi kategoriaksi. Tutkimuksen kannalta oli oleel-

lista lisätä kaksi objektiivista kategoriaa: Käyttöönotto CI/CD implementaatioissa ja Haavoittuvuuksien tulkinta. Näiden kategorioiden lisäys perustui tarpeeseen arvioida haavoittuvuuksien tulkinnan helppoutta sekä työkalujen integrointia kehitysputkeen. Lokitukseen ja hallintaan, heikkoon ylläpitoon sekä resurssirajoitteisiin liittyvät uhkat jätettiin pois kategorisoinnista, ja perustelut tälle valinnalle esitetään luvussa 6.3.

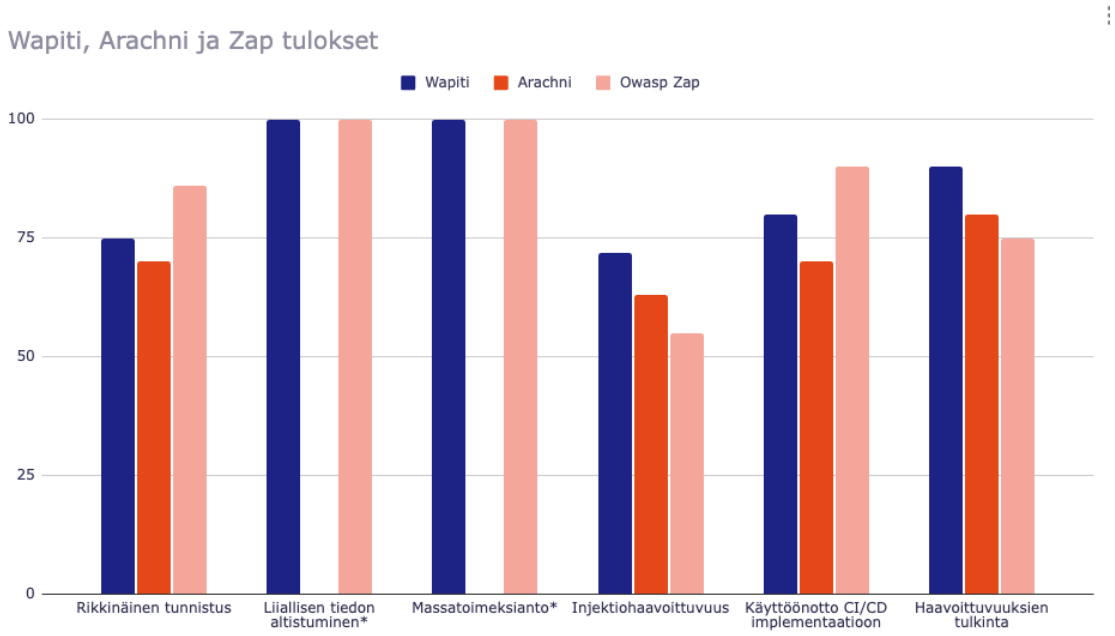
6.2 Testien tulokset

Tässä osiossa käsitellään tuloksia DAST-, SAST- ja IAST-järjestelmien välillä ja vertaillaan järjestelmiä keskenään. Tulokset on ryhmitelty ohjelmatyyppien mukaan johtuen niiden teknisistä erityispiirteistä, jotka on yksityiskohtaisesti kuvattu luvussa 2. Tämä erottelu on tärkeä, sillä se mahdollistaa syvällisemmän ymmärryksen kunkin järjestelmätyypin tehokkuudesta ja soveltuvuudesta.

6.2.1 DAST-järjestelmien tulokset

Tutkimuksessa testattiin OWASP ZAP, Arachni Scanner ja Wapiti -järjestelmiä. Testausprosessit olivat samankaltaisia: jokainen järjestelmä aloitti toimintansa etsimällä päätepisteitä Benchmark-ohjelmasta. Löydettyään päätepisteet järjestelmät suorittivat niissä testejä. Testien suoritusajassa havaittiin, että Arachni Scanner -järjestelmän testiajo kesti tunnin kauemmin kuin Wapitin ja ZAP:n testiajot. CI/CD-putkessa suoritusajan pituus ei ole kriittinen, sillä useita testejä voidaan suorittaa rinnakkain ilman, että putki jumiutuu. Suoritusajan pituus on kuitenkin tekijä, joka tulee ottaa huomioon CI/CD-putkea suunniteltaessa.

Kuvassa 6.1 esitetään DAST-järjestelmien tulokset. Tulokset vaihtelivat merkittävästi kategorioiden(Rikkinäinen objektitunnistus, Liiallisen datan altistaminen, Massatoimeksianto ja Injektio) välillä. Wapiti ja ZAP saavuttivat parhaat tulok-



Kuva 6.1: Tulosten vertailu DAST-järjestelmien välillä.

set luvussa 3 esitellyissä kategorioissa, kun taas Arachni saavutti heikkoja tuloksia näissä kategorioissa mutta menestyi hyvin tutkimuksen ulkopuolelle jätetyissä kategorioissa. Injektio-kategoriassa havaittiin suurimmat vaihtelut järjestelmien välillä, mikä johtui injektiotestauksen erilaisista toteutuksista ja alakategorioista, kuten LDAP- ja SQL-injektioista.

Liiallisen datan altistamisen ja massatoimeksiannon kategorioissa vaadittiin lisäkonfiguraatiota, jotta tuloksia saatiin. Ilman tätä konfiguraatiota järjestelmät eivät saaneet pisteitä. Haavoittuvuuksien tunnistamiseksi järjestelmät vaativat avainlistan. Arachni Scanner ei tukenut näitä kategorioita, joten se ei tuottanut tuloksia niissä.

Kahdessa viimeisessä kategoriassa suoritettiin vertailu, joka käsitteli ohjelmien käyttöönottoa ja löydettyjen haavoittuvuuksien tulkintaa. Kaikkien järjestelmien todettiin toimivan samankaltaisesti käyttöönoton osalta. Ne kaikki tarjosivat useita vaihtoehtoja ohjelman suorittamiseen, kuten virtuaalikonepohjaisia ratkaisuja

CI/CD-putkeen. ZAP-järjestelmä erottui muista tarjoamalla parhaan käyttökokemuksen ja vaatimalla vähiten konfiguraatiota täyden toiminnallisuuden saavuttamiseksi. Arachni ja Wapiti vaativat ZAP-järjestelmää enemmän asetusten säätämistä toimiakseen kattavasti. Arachnin heikkoa käyttökokemusta voitaisiin selittää sillä, ettei siihen päivityslokien perusteella ole tehty päivityksiä moniin vuosiin.

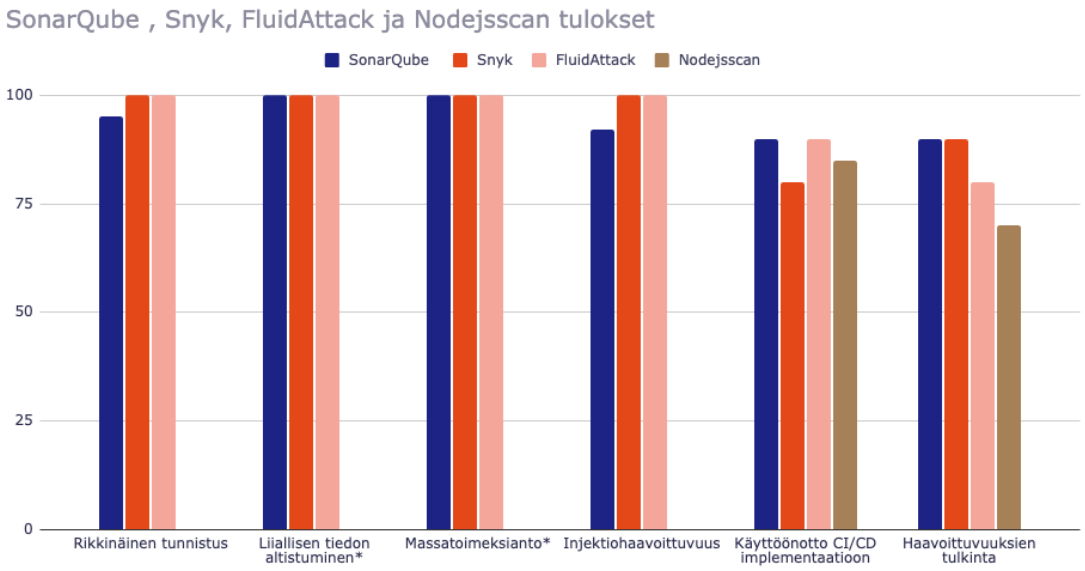
Haavoittuvuuksien tulkinnassa Wapiti suoriutui parhaiten esittäessään selkeästi uhkan tason, tunnistusmenetelmän ja sen tapahtumapaikan. Tämä selkeys auttaa korjaustoimenpiteiden tehokkaassa testaamisessa. ZAP ja Arachni tarjosivat keskenään verrattain samankaltaisia haavoittuvuuksien kuvauksia, jotka sisälsivät yleistason tiedot ja CWE-viitteet. ZAP kuitenkin tarjosi laajemman valikoiman luokkia ja kategorioita haavoittuvuuksille, kuten informatiiviset ja väärän hälytyksen kategoriat, mikä teki sen tulosten tulkinnasta monimutkaisempaa.

OWASP ZAP, Arachni Scanner ja Wapiti -haavoittuvuusanalyysijärjestelmien vertailussa havaittiin, että vaikka kaikki kolme järjestelmää suorittavat perustoiminnot samankaltaisesti, niiden välillä on merkittäviä eroja suoritusnopeudessa, haavoittuvuuksien kategorisoinnissa ja käyttökokemuksessa. Arachni Scanner osoittautui hitaimmaksi, mikä voi vaikuttaa CI/CD-putken suunnitteluun. Haavoittuvuuksien tunnistamisen tarkkuudessa Wapiti ja ZAP saavuttivat parhaat tulokset tietyissä haavoittuvuuskategorioissa, kun taas Arachni erottui muissa kategorioissa, joita tutkimuksessa ei erikseen käsitelty. Käyttökokemuksen osalta ZAP erottui edukseen tarjoamalla parhaan käyttökokemuksen vähäisen konfiguraation tarpeen myötä. Wapiti tarjosi selkeimmän tiedon haavoittuvuuksien tulkinnassa, mikä on tärkeää korjaustoimenpiteiden validoinnissa.

6.2.2 SAST-järjestelmien tulokset

SAST-järjestelmistä testauksessa olivat mukana SonarQube, FluidAttack, Snyk ja Nodejsscan. Niiden toiminta perustui lähdekoodin tarkasteluun. SonarQube, Flui-

dAttack ja Nodejsscan suoritettiin virtuaalikoneelta, kun taas Snyk käytettiin GitHubin kautta, kuten luvussa 4.2.4 on kuvattu. Kaikki järjestelmät osoittautuivat yhteensopiviksi CI/CD-putkien erilaisiin toteutuksiin.



Kuva 6.2: Tulosten vertailu SAST-järjestelmien välillä.

SAST-järjestelmien tulokset esitellään kuvassa 6.2. SonarQube, Snyk ja FluidAttack tuottivat keskenään samankaltaisia tuloksia. Nodejsscan erottui tuloksissa merkittävästi muiden järjestelmien tuloksista, mikä johtui sen rajoittuneesta tuesta vain JavaScriptille ja Node.js-teknologialle. Tämä rajoitus aiheutti sen, että Nodejsscan ei kyennyt tuottamaan merkityksellisiä tuloksia Benchmark-ohjelmassa.

SonarQuben, Snykin ja FluidAttackin välinen vertailu osoittautui tasaväkiseksi. SonarQube generoi useita hälytyksiä, jotka vaihtelivat aidoista haavoittuvuuksista tyylillisiin seikkoihin ja sisälsivät myös muutamia vääriä positiivisia hälytyksiä. FluidAttack ja Snyk puolestaan loivat virheettömiä raportteja ja tunnistivat kaikki haavoittuvuudet ilman vääriä osumia.

CI/CD-implementaation arviointi osoittautui haastavaksi avoimen lähdekoodin

lisenssien vuoksi. FluidAttack, SonarQube ja Nodejsscan tarjosivat monipuolisia implementointivaihtoehtoja, kuten keskitettyjä palvelinratkaisuja ja virtualisointimahdollisuuksia, sekä versionhallintajärjestelmien integraatiota. Tämä mahdollisti lähdekoodin integroinnin sisäverkkoon, mikä tekisi sensitiivisen koodin tarkastamisen mahdolliseksi työkaluilla. Ilmaisena versiona Snyk tarjosi vain GitHub-integraation, mutta kaupallinen versio sisälsivastaavanlaisia ominaisuuksia kuin muutkin järjestelmät. SonarQube ja FluidAttack saivat tästä syystä paremmat arviot, sillä ne tarjosivat perusteelliset ohjeistukset implementaatioon.

Haavoittuvuuksien tulkinta -kategoriassa havaittiin yhdenmukaisuutta kaikissa SAST- järjestelmissä. Järjestelmät raportoivat haavoittuvuudet samanmuotoisesti, ilmoittivat haavoittuvuuden sijainnin koodissa ja kertoivat yleiskuvauksen haavoittuvuudesta. SonarQube ja Snyk tarjosivat kattavan CWE-kuvauksen ja yleisiä korjausehdotuksia haavoittuvuuksille, kun taas Nodejsscan ja FluidAttack antoivat haavoittuvuuksille yleiset nimet. Snykin ja Sonar-Quben tarjoamien korjausehdotusten on todettu olevan yleisiä, ja ne eivät aina soveltuneet erityiseen kohdesovellukseen, mutta ne toimivat ohjeistuksena kehittäjille.

Testauksessa ilmeni, että SonarQuben ja Snykin tuottamiin koodin tyylillisiin parannusehdotuksiin liittyy tulosten tulkintaa hankaloittava haaste: ne sekoittuivat haavoittuvuusilmoituksiin ja vaikeuttivat tulosten arviointia. Nämä tyylilliset ehdotukset eivät olleet poiskytkettävissä. Toisaalta FluidAttackin raportti oli yksinkertaisempi, sillä siinä ei ollut korjausehdotuksia eikä koodin tyyliseikkoihin puututtu. Nodejsscanin raportti toimi tehokkaasti Node.js-tekniikkaa käyttävissä projekteissa, sillä se tarjosi korjausehdotuksia vastaavalla tavalla kuin Snyk ja SonarQube. Kuitenkin, kuten aiemmin todettiin, Nodejsscan ei kyennyt tuottamaan pätevää raporttia Benchmark-ohjelman haavoittuvuuksista.

SAST-järjestelmien vertailussa SonarQube, Snyk ja FluidAttack saavuttivat johdonmukaisesti korkeita tuloksia kaikissa testatuissa kategorioissa, mikä osoittaa nii-

den kyvyn tunnistaa laaja kirjo haavoittuvuuksia eri teknologiaympäristöissä. Nodejsscan puolestaan tuotti muita järjestelmiä alhaisempia pisteitä, mikä korostaa sen erikoistumista Node.js-ympäristöihin. SonarQube loisti tuottaessaan suuren määrän hälytyksiä mutta kärsi väärien positiivisten tulosten määrästä, kun taas FluidAttack ja Snyk erottuivat virheettömillä raporteillaan. CI/CD-integraation osalta SonarQube ja FluidAttack tarjosivat laajemmat implementointivaihtoehdot ja kattavamman dokumentaation verrattuna Snykin yksinkertaisempaan GitHub-integraatioon. SonarQube ja Snyk tarjosivat lisäarvoa tarkoilla CWE-kuvauksilla ja korjausehdotuksilla.

6.2.3 IAST-järjestelmien tulokset

Tässä tutkielmassa IAST-järjestelmiä ei voitu käytännössä kokeilla, sillä yhtäkään avoimen lähdekoodin järjestelmää ei tutkimushetkellä ollut saatavilla. Seeker- ja Hdiv-järjestelmien kohdalla löytyi kuitenkin dokumentaatiota niiden testauksesta Benchmark-järjestelmää vastaan [40][4]. Dokumenttien perusteella ne saivat täydet pisteet SAST- ja DAST-kategorioissa. Tämä oli oletettua, sillä Seeker ja Hdiv pystyvät hyödyntämään molempien testien analyyseja. Acunetixin ja Netsparkerin testauksesta Benchmark-ohjelmistoa vastaan ei löytynyt dokumentaatiota. Niistä löytyi ainoastaan dokumentteja testiajoista yrityksien itse tuottamiin projekteihin. Niitä ei voitu käyttää vertailukohteena.

6.3 Testien ulkopuolelle jääneet haavoittuvuudet

Alun perin luvussa 3 esiteltyjen rajapintojen haavoittuvuudet eivät ole täysin kattavia tässä tutkimuksessa käsiteltyjen järjestelmien osalta. Puutteellinen monitorointi, ylläpito ja resurssienhallinnan valvonta sekä virheelliset tietoturvan määritykset jäivät tutkimuksen ulkopuolelle, vaikka ne ovat keskeisiä haavoittuvuuksia. On

havaittu, että SAST- ja DAST-järjestelmät eivät kykene havaitsemaan mainittuja haavoittuvuuksia. Näiden haavoittuvuuksien huomioon ottaminen automatisoidun tietoturvan suunnittelussa rajapintakehityksessä esitellään tässä alaluvussa.

Automatisaation avulla voidaan puuttua puutteelliseen resurssien hallintaan ja monitorointiin. Tässä yhteydessä korostuu suoritusympäristön merkitys. Virtualisointi, joka on keskeinen osa CI/CD-putkea ja ohjelmistokehitystä, mahdollistaa joustavan resurssienhallinnan. Monitorointi on toteutettavissa alustasta riippumatta, ja esimerkiksi RASP-tekniikka mahdollistaa lokituksen sekä resurssienhallinnan automatisointia, kuten luvussa 2.6 mainitaan.

Ylläpidon ja tietoturvamääritysten osalta automaattiset järjestelmät osoittautuivat riittämättömiksi, kuten tulokset luvuissa 6.1 ja 6.2 todistavat. Ylläpidon integrointi ohjelmoitavan rajapinnan suunnitteluun on välttämätöntä, mukaan lukien vastuuhenkilöiden määrittely ja teknisen toteutuksen huolellinen dokumentointi. Dokumentaation avulla ylläpito helpottuu, ja se mahdollistaa systemaattisen lähestymistavan ongelmatilanteiden korjaamiseen.

Projektikohtaiset tietoturvamääritykset määräytyvät projektin tuottajan ja ti laajan vaatimusten mukaan. Tuottajan on varmistettava tietoturvamääritysten toteutuminen, ja niiden noudattamista voidaan valvoa tietoturva-arvioinnein. Tieto turvamääritysten haasteena on niiden moninaisuus ja kriteerien vaihtelu tilaajan mukaan.

6.4 Tulosten yhteenveto

Tutkimuksen tulokset osoittavat, että SAST-järjestelmät ovat erityisen tehokkaita tunnistamaan haavoittuvuuksia ohjelmistoista, erityisesti rajapintakehityksen kontekstissa. Niiden avulla on mahdollista löytää merkittäviä haavoittuvuuksia suoraan ohjelmiston lähdekoodista. DAST-järjestelmät puolestaan tarjoavat näkemyksen hyökkääjän perspektiivistä, mikä mahdollistaa sellaisten ongelmien paljastami-

sen, joita SAST-järjestelmät eivät välttämättä havaitse. Näihin ongelmiin kuuluvat esimerkiksi ulkopuolelle näkyvät päätepisteet, jotka eivät saisi olla näkyvissä. Tästä huolimatta DAST-järjestelmien tarjoama analyysi on SAST-järjestelmien analyysia rajallisempi. Ne voivat ilmoittaa haavoittuvan päätepisteen ja kutsun, mutta eivät tar-joa tarkkaa tietoa haavoittuvuuden sijainnista koodissa. SAST-järjestelmien vertai-lussa FluidAttack ja Snyk erottuivat suorituskyvyltään parhaimpina, kun taas DAST-järjestelmistä Wapiti ja ZAP osoittautuivat tehokkaimmiksi.

7 Pohdinta

Tässä luvussa arvioidaan tutkimuksen tuloksia viitekehyksen kontekstissa. Pohditaan tutkimuksen rajoituksia ja annetaan ehdotuksia jatkotutkimukselle.

7.1 Tulosten pohdinta

Tämän tutkimuksen testeissä havaittiin, että useimmat SAST- ja DAST-järjestelmät suoriutuivat hyvin. On todettu, että automaattiset tietoturvan järjestelmät kykenevät havaitsemaan ohjelmoitavien rajapintojen haavoittuvuuksia. Tutkitut järjestelmät tunnistivat ohjelmoitavien rajapintojen haavoittuvuudet, jotka on esitetty luvussa 3.

SAST-järjestelmien DAST-järjestelmiin verrattuna parempi suoriutuminen johtunee niiden kyvystä analysoida haavoittuvuuksia lähdekoodista. Toisaalta DAST-järjestelmät toteuttavat testauksen tavalla, joka muistuttaa ulkopuolisen hyökkääjän toimintaa paljastaen haavoittuvuuksia eri menetelmällä kuin lähdekoodianalyysi. Molemmat järjestelmätyypit tarjoavat erilaisen näkökulman haavoittuvuuksien tunnistamiseen.

Järjestelmän kyky tunnistaa haavoittuvuuksia ei yksinään määrittele sen tehokkuutta. On tärkeää, että järjestelmä esittää tulokset käyttäjälle selkeästi ymmärrettävässä muodossa. Testatuissa järjestelmissä oli eroja tulosten esitystavassa. Tulosten selkeys on subjektiivinen ja riippuu käyttäjän näkemyksestä. Järjestelmää valittaessa tulisi ottaa huomioon käyttäjäperspektiivi. Jotkin järjestelmät tarjoavat

laajasti tietoa ja luokituksia, mikä voi olla haastavaa aiheeseen perehtymättömälle.

Kaikki tutkimuksessa testatut järjestelmät mahdollistivat integraation CI/CD -putkeen. On merkittävää, että useat järjestelmät tarjoavat kaupallisen version, joka sisältää laajemman tukipalvelun ja lisäominaisuuksia. Ilmaisversiot tarjosivat rajoitettuja integraatiomahdollisuuksia, kun taas täysin avoimen lähdekoodin järjestelmät esittivät laajempia ratkaisuja, mutta vaativat käyttäjältä suurempaa osaamista. CI/CD-putken integrointivaatimukset ja käytettävissä olevat resurssit tulisi ottaa huomioon järjestelmää valittaessa. Ratkaisevia tekijöitä ovat taloudelliset resurssit, aika, projektin laatu ja kehittäjän tietotaito.

Erilaiset järjestelmät tarjoavat erilaisia toimintamalleja, mikä tarkoittaa, että yksittäinen järjestelmä ei ole paras kaikissa tilanteissa. Tämän ovat myös todenneet Mburano et al. artikkelissaan [5]. Optimaalinen automatisoidun rajapintaturvallisuuden malli vaatisi useamman SAST-, DAST- ja IAST-järjestelmän käyttöönoton. Kuten luvussa 2 esitetään, järjestelmät tarjoavat erilaisia etuja ja toimivat eri tavoin. CI/CD-putken avulla voidaan hyödyntää useita tietoturvatestausjärjestelmiä, mikä on suositeltavaa. SAST- ja DAST-järjestelmien yhdistelmällä voidaan tunnistaa haavoittuvuuksia sekä lähdekoodista että ulkopuolisen hyökkääjän perspektiivistä ja tarjota kattava yleiskuva rajapinnan turvallisuudesta. Tämän tutkimuksen perusteella voidaan suositella SonarQubea SAST-järjestelmänä ja OWASP ZAPia DAST-järjestelmänä.

Monitoroinnin, ylläpidon, tietoturvamääritysten ja resurssienhallinnan merkitys applikaation tuotantoympäristössä ja suunnitteluvaiheessa on olennainen. Nämä tekijät riippuvat projektista, sen tuottajasta ja tilaajasta. Monitoroinnin, ylläpidon, tietoturvamääritysten ja resurssienhallinnan suunnittelu tulisi sisällyttää järjestelmän suunnitteludokumentaatioon. Tällöin varmistetaan, että kaikki tietoturvaan liittyvät osa-alueet kattavaan rajapinnan kehittämiseen otetaan huomioon tehokkaasti,ja mahdollistetaan paras mahdollinen tieto-turva rajapintajärjestelmälle.

7.2 Luotettavuus ja eettisyys

Automatisoitua tietoturvaa käsittelevän lähdemateriaalin haasteeksi on tunnistettu puolueettoman, eri järjestelmiä vertailevan tutkimuksen puuttuminen. Tämän tutkimuksen luotettavuutta vahvistaa sen puolueettomuus; tuloksiin ei ole vaikuttanut esiteltyjen ohjelmistojen tarjoajat eikä mikään ulkoinen taho. Tutkimuksen luotettavuutta korostaa edelleen se seikka, että kaikki järjestelmät on testattu identtisillä metodeilla yhden henkilön toimesta. Tutkijan opiskelutausta ja tietoturvaan kohdistuva harrastuneisuus ovat myös vaikuttaneet positiivisesti tulosten tulkintaan.

Tämänhetkistä markkinatilannetta edustavan laajan otannan ansiosta tutkimus on merkittävä. Otanta koostuu tieteellisissä julkaisuissa mainituista järjestelmistä sekä alan muusta kirjallisuudesta löytyvistä järjestelmistä. Markkinatilanteen muuttuvuus on huomioitu: uusia järjestelmiä kehitetään, vanhoja parannetaan ja joitakin järjestelmien ylläpitoja lopetetaan. Esimerkkinä muutoksesta on Github Advanced Security tool, jonka toiminta keskeytettiin ja käynnistettiin uudelleen uudella nimellä tutkimuksen aikana.

Tutkimuksen luotettavuuteen kohdistuva haaste on Benchmark-järjestelmän toteutus, joka käyttää vanhempia tekniikoita eikä vastaa nykyisiä toteutuksia. Tämä vaikuttaa testien toistettavuuteen ja erityisesti tulevaisuuden vertailukelpoisten tulosten tuottamiseen. Luotettavuuden parantamiseksi tulevaisuudessa olisi kehitettävä parempi automatisoitujen tietoturva-arviointien järjestelmä. Benchmark-järjestelmä on tällä hetkellä ainoa, joka pysyy ajan tasalla haavoittuvuuksien osalta.

7.3 Jatkotutkimusehdotukset

Tämän tutkimuksen tulosten pohjalta identifioidaan kaksi jatkotutkimuksen aluetta. Ensimmäisenä ehdotetaan, että tutkitaan esimerkkitapauksen kehittämistä: Tässä automatisoitua tietoturvaa hyödynnetään rajapintaohjelmiston kehittämisessä.

Käytössä ovat sekä SAST- että DAST-järjestelmät yhdistettynä tunnistamattomien haavoittuvuuksien turvaamiseen dokumentaation avulla. Tutkittavaksi ehdotetaan kehittäjille ja yritykselle koituvia etuja ja haittoja.

Toisena jatkotutkimuksen alueena ehdotetaan yhtenäisen testausjärjestelmän luomista. Järjestelmä mahdollistaisi SAST-, DAST- ja IAST-järjestelmien testaamisen yhdessä ympäristössä. Nykyiset järjestelmät perustuvat vanhentuneisiin tekniikoihin, eivätkä ne kata uusimpia ja yleisimpiä haavoittuvuuksia. Standardoitu testausprosessi parantaisi ohjelmistojen välistä läpinäkyvyyttä ja vertailukelpoisuutta. Lisäksi ehdotetaan testausjärjestelmän käytettävyyden ja ketterän kehityksen yhteydessä tapahtuvan implementoinnin tutkimista.

Lähdeluettelo

[1] F. Hussain, R. Hussain, B. Noye ja S. Sharieh, "Enterprise API Security and GDPR Compliance: Design and Implementation Perspective", *IT Professional*, vol. 22, s. 81–89, 2020. DOI: 10.1109/MITP.2020.2973852.

[2] H. Zhong ja X. Wang, "An empirical study on API usages from code search engine and local library", *Empirical Software Engineering*, vol. 28, s. 63, 2023. DOI: 10.1007/s10664-023-10304-z.

[3] Y. Stefinko, A. Piskozub ja R. Banakh, "Manual and automated penetration testing. Benefits and drawbacks. Modern tendency", teoksessa *2016 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*, Lviv, Ukraine, 2016, s. 488–491. DOI: 10.1109/TCSET.2016.7452095.

[4] Synopsis, *Seeker Interactive Application Security Testing*, 2022. url: https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/interactive-application-security-testing-datasheet.pdf, Luettu 1.9.2022.

[5] B. Mburano ja W. Si, "Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark", teoksessa *2018 26th International Conference on Systems Engineering (ICSEng)*, Sydney, NSW, Australia, 2018, s. 1–6. DOI: 10.1109/ICSENG.2018.8638176.

[6] D. Gupta, K. Chatterjee ja S. Jaiswal, "A Framework for Security Testing", teoksessa *Computational Science and Its Applications – ICCSA 2013*, B. Murgante, S. Misra, M. Carlini, C. M. Torre, H.-Q. Nguyen, D. Taniar, B. O. Apduhan ja O. Gervasi, toim., Ho Chi Minh City, Vietnam, 2013, s. 187–198. DOI: 10.1007/978-3-642-39646-5_14.

[7] T.-y. Gu, Y.-s. Shi ja Y.-y. Fang, "Research on Software Security Testing", *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 4, s. 1446–1450, 2010. DOI: 10.5281/zenodo.1081388.

[8] M. Felderer, B. Agreiter, P. Zech ja R. Breu, "A classification for model-based security testing", *Advances in System Testing and Validation Lifecycle (VALID 2011)*, s. 109–114, 2011.

[9] I. Schieferdecker, J. Grossmann ja M. Schneider, "Model-Based Security Testing", *Electronic Proceedings in Theoretical Computer Science*, vol. 80, s. 1–12, 2012. DOI: 10.4204/eptcs.80.1.

[10] Y. Ongena, *Creating a CI/CD pipeline between Jenkins and Mobile Cloud Services*, 2021. url: https://www.ateam-oracle.com/creating-a-cicd-pipeline-between-jenkins-and-mobile-cloud-services, Luettu 20.8.2022.

[11] K. Klooster, "Applying a Security Testing Methodology: a Case Study", Bachelor's thesis, University of Tartu, 2016. url: https://core.ac.uk/download/pdf/83597588.pdf.

[12] M. R. Albrecht ja R. B. Jensen, "The Vacuity of the Open Source Security Testing Methodology Manual", teoksessa *Security Standardisation Research*, vol. 12529, London, UK, 2020, s. 114–147. DOI: 10.1007/978-3-030-64357-7_6.

[13] Y. Pan, "Interactive Application Security Testing", teoksessa *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, Xiangtan, China, 2019, s. 558–561. DOI: 10.1109/ICSGEA.2019.00131.

[14] J. Yang, L. Tan, J. Peyton ja K. A. Duer, "Towards Better Utilizing Static Application Security Testing", teoksessa *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Montreal, QC, Canada, 2019, s. 51–60. DOI: 10.1109/ICSE-SEIP.2019.00014.

[15] J. Li, "Vulnerabilities Mapping based on OWASP-SANS: a Survey for Static Application Security Testing (SAST)", *Annals of Emerging Technologies in Computing*, vol. 4, s. 1–8, 2020. DOI: 10.48550/arXiv.2004.03216.

[16] T. Rangnau, R. Buijtenen, F. Fransen ja F. Turkmen, "Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines", teoksessa *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, Eindhoven, Netherlands, 2020, s. 145–154. DOI: 10.1109/EDOC49727.2020.00026.

[17] H. Setiawan, L. E. Erlangga ja I. Baskoro, "Vulnerability Analysis Using The Interactive Application Security Testing (IAST) Approach For Government X Website Applications", teoksessa *2020 3rd International Conference on Information and Communications Technology (ICOIACT)*, Yogyakarta, Indonesia, 2020, s. 471–475. DOI: 10.1109/ICOIACT50329.2020.9332116.

[18] R. Piyush, *What Runtime Application Self-Protection (RASP) Doesn't Solve*, 2021. url: https://www.traceable.ai/blog-post/what-runtime-application-self-protection-rasp-doesnt-solve, Luettu 15.9.2022.

[19] Z.-N. Chen, K. Chen, J. Jiang, L.-F. Zhang, S. Wu, Z. Qi, C.-M. Hu, Y.-W. Wu, Y. Sun, H. Tang, A.-B. Sun ja Z.-L. Kang, "Evolution of Cloud Operating

System: From Technology to Ecosystem", *Journal of Computer Science and Technology*, vol. 32, s. 224–241, 2017. DOI: 10.1007/s11390-017-1717-z.

[20] M. Nehra, *Types of APIs | What are APIs? | Different types of APIs*, 2019. url: https://dev.to/decipherzonesoft/types-of-apis-what-are-apis-different-types-of-apis-3mjm, Luettu 1.6.2022.

[21] A. Munsch PhD ja P. Munsch MBA, "The Future of API (Application Programming Interface) Security: The Adoption of APIs for Digital Communications and the Implications for Cyber Security Vulnerabilities", *Journal of International Technology and Information Management*, vol. 29, s. 24–45, 2021. DOI: 10.58729/1941-6679.1454.

[22] OWASP, *OWASP: API Security TOP 10 2019*, 2019. url: https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf, Luettu: 2022-05-28.

[23] L. Tang, L. Ouyang ja W.-T. Tsai, "Multi-factor web API security for securing Mobile Cloud", teoksessa *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, Zhangjiajie, China, 2015, s. 2163–2168. DOI: 10.1109/FSKD.2015.7382287.

[24] A. Viriya ja Y. Muliono, "Peeking and Testing Broken Object Level Authorization Vulnerability onto E-Commerce and E-Banking Mobile Applications", *Procedia Computer Science*, vol. 179, s. 962–965, 2021. DOI: 10.1016/j.procs.2021.01.101.

[25] E. G. Demesa, "Implementation of a Hands-on Attack and Defense Lab on Insecure Direct Object References", Master's thesis, Tallinn University of Technology School of Information Technologies, 2019. url: https://www.etis.ee/Portal/Mentorships/Display/57461a3c-f3aa-40f8-a9e3-05a76e074551.

[26] A. Greenberg, *An Absurdly Basic Bug Let Anyone Grab All of Parler's Data*, 2021. url: https://www.wired.com/story/parler-hack-data-public-posts-images-video/, Luettu 1.7.2022.

[27] A. Bisegna, R. Carbone, M. Ceccato, S. Manfredi, S. Ranise, G. Sciarretta, A. Tomasi ja E. Viglianisi, "6. Automated Assistance to the Security Assessment of API for Financial Services", teoksessa *Cyber-Physical Threat Intelligence for Critical Infrastructures Security: A Guide to Integrated Cyber-Physical Protection of Modern Critical Infrastructures*. 2020, s. 94–103. DOI: 10.1561/9781680836875.ch6.

[28] J. Michener, "Security Issues With Functions as a Service", *IT Professional*, vol. 22, s. 24–31, 2020. DOI: 10.1109/MITP.2019.2930049.

[29] D. Ray ja J. Ligatti, "Defining code-injection attacks", *Acm Sigplan Notices*, vol. 47, s. 179–190, 2012. DOI: 10.1145/2103621.2103678.

[30] OWASP, *OWASP ZAP*, 2011. url: https://www.zaproxy.org/docs/, Luettu 22.8.2022.

[31] Arachni-Scanner, *Arachni Web application security scanner framework*, 2018. url: https://www.arachni-scanner.com, Luettu 30.8.2022.

[32] Wapiti, *Wapiti: The web-application vulnerability scanner*, 2019. url: https://wapiti.sourceforge.io, Luettu 28.8.2022.

[33] Invicti, *Changing the DAST Game with Netsparker IAST*, 2020. url: https://www.netsparker.com/changing-the-dast-game-with-netsparker-iast-white-paper/, Luettu 30.8.2022.

[34] Invicti, *Spring 2021 Edition: Acunetix Web Vulnerability Report*, 2021. url: https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2021/, Luettu 30.8.2022.

[35] Sonarqube, *sonarqube.org*, 2018. url: https://www.sonarqube.org, Luettu 23.8.2022.

[36] F. Attacks, *fluidattacks.com*, 2019. url: https://fluidattacks.com, Luettu 23.8.2022.

[37] A. Abraham, *Nodejsscan*, 2017. url: https://github.com/ajinabraham/nodejsscan, Luettu 1.9.2022.

[38] Snyk.io, *What is Snyk?*, 2015. url: https://snyk.io/what-is-snyk/, Luettu 1.9.2022.

[39] Kiuwan, *The OWASP Benchmark & Kiuwan*, 2017. url: https://www.kiuwan.com/owasp-benchmark-kiuwan/, Luettu 1.9.2022.

[40] Hdiv, *OWASP Benchmark Hdiv Detection (IAST)*, 2020. url: https://hdivsecurity.com/owasp-benchmark, Luettu 1.9.2022.

[41] Wavesep, *The Web Application Vulnerability Scanner Evaluation Project*, 2014. url: https://github.com/sectooladdict/wavsep, Luettu 30.8.2022.

[42] OWASP, *OWASP Benchmark*, 2019. url: https://owasp.org/www-project-benchmark/, Luettu 20.8.2022.