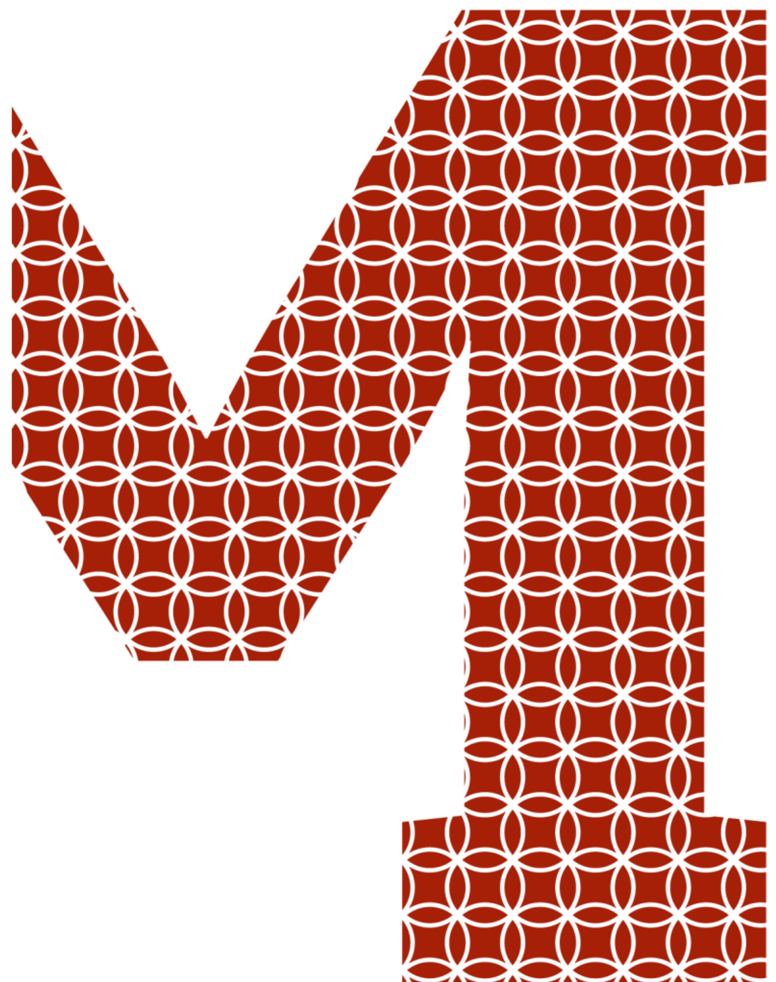


Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Otso Pohjola

Tietoturvaraporttien yhteenmuotoilu

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

25.4.2019

Tekijä Otsikko Sivumäärä Aika	Otso Pohjola Tietoturvaraporttien yhteenmuotoilu 50 sivua + 1 liite 25.4.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	yliopettaja Erja Nikunen tietoturva-asiantuntija Tommi Kananoja

Insinöörityön aiheena oli tietoturvatyökalujen raporttien standardointi. Työ tehtiin olemassa olevaan sovellustietoturvaprojektiin Accenturelle, jossa tarvittiin tapa käsitellä eri tietoturvaraporttien tietoja yhdessä järjestelmässä. Työn ohessa oli tarkoitus tutustua erilaisiin tietoturvaskannereihin sekä niiden raportteihin, ja se toteutettiin Django-nimisellä Python-ohjelmistokehyksellä.

Työhön valitut skannerit edustavat eri tapoja tehdä sovellustietoturvaan liittyviä testejä. Kun staattiset työkalut SonarQube, DependencyCheck ja Retire.js etsivät lähdekoodista haavoittuvuuksia, Zed Attack Proxy, OpenVAS ja Retina taas testaavat käynnissä olevaa ohjelmaa muun muassa simuloimalla hyökkäyksiä. Raportteja tarkastellessa huomattiin kuitenkin niiden olevan keskenään sisällöllisesti hyvin samanlaisia. Yhteisen rakenteen suunnittelussa päädyttiin laittamaan havaintotiedot yhteen luokkaan tyypitettyihin aliluokkiin jakamisen sijaan, jotta ohjelman jatkokehitys pysyisi mutkattomana.

Jäsentimiä varten luotiin abstrakti luokka, joka toteuttaa niiden tietojen tallentamisen ja raporttien esikäsittelyn. Siitä periytyy jokaiselle työn tietoturvaskannerille oma käsittelijänsä, jonka toteutus vastaa aina kunkin raportin muotoa ja sisältöä. Kolmelle eri skannerille nämä koodattiin kahdella eri tavalla, joista pyrittiin valitsemaan parempi vertailemalla eri toteutustapojen ajan- ja muistinkäyttöä. Ensimmäinen tapa perustui tavallisiin for-silmukoihin, ja jälkimmäinen Pythonin generaattoreihin. Generaattoritoteutus päätettiin ottaa käyttöön projektissa sen vähän paremman suorituskyvyn takia.

Kehitettyjä jäsentimiä ja rakenteita tullaan käyttämään havaintojen visualisointiin ja hallinnointiin. Myös havaintojen kaksoiskappaleiden poistamiseen sekä uusien työkalujen liittämiseen tullaan paneutumaan jatkokehityksessä.

Avainsanat	sovellustietoturva, silmukat, kyberturvallisuus, Python

Author Title Number of Pages Date	Otso Pohjola Unifying security reports' formats 50 pages + 1 appendices 25 April 2017
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Erja Nikunen, Principal Lecturer Tommi Kananoja, Security Specialist

The subject of this bachelor's thesis was standardization of security testing reports. It is a part of Accenture's ongoing application security project which needed a way to unite security reports originating from various cyber security testing tools. This required learning about them and their reports' structure and content. The thesis was implemented on a Python based framework called Django.

The project's security testing tools represent different ways to commit application security related testing. While SonarQube, Dependency-Check and Retire.js are static tools and look for vulnerabilities in the source code, there are also dynamic tools called Zed Attack Proxy, OpenVAS and Retina that test the target dynamically. They require the target software to be running and can do testing by simulating real attacks. Despite these differences, all the reports are more or less similar in case of their contents.

While planning the common structure, all the data related to findings were put into a single class so that the further development would be simpler. Also, an abstract class was created for the parsers. It takes care of saving and preprocessing the reports, and it's inherited by the implemented parsers. They are made for each security scanner and their implementation follows each's content and layout.

There were two versions of the parser implementation for three scanners, which's time and memory efficiency were compared. The first was based on basic for-loops and the second used Python's generators. The latter was considered to be slightly superior and was chosen to be used in the project.

Parsers and structures developed in this project will be made use of in the future when visualizing and managing findings. There will also be some focus on removing duplicate findings and adding additional cyber security scanners.

Keywords	application security, loops, cybersecurity, Python

Sisällys

Lyhenteet

1 Johdanto 1

2 Taustatutkimus 3

3 Insinöörityön taustatietoja 4

3.1 Django 4
3.2 Havainto ja haavoittuvuus 5

4 Tietoturvaskannerit 9

4.1 SonarQube 10
4.2 Riippuvuuksia testaavat SAST-skannerit 13
4.3 Zed Attack Proxy 14
4.4 OpenVAS 16
4.5 Retina 19

5 Raportit 21

5.1 Common vulnerability scoring system (CVSS) 21
5.2 SonarQube 22
5.3 Retire.js 22
5.4 Dependency-Check 23
5.5 Zed Attack Proxy 24
5.6 OpenVAS 25
5.7 Retina 27

6 Rakenne 28

7 Jäsentimet 32

7.1 Listatoteutus ja tallentaminen 34
7.2 Generaattoritoteutus 37
7.3 Suorituskykyvertailu 39

8 Yhteenveto 44

Lähteet 45

Liitteet

Liite 1. Raporttitietojen rakenteen luokkakaaviot (poistettu julkisesta versiosta salaisena)

Lyhenteet

BIOS	Basic input/output system. Tietokoneen mikroprosessorin käyttämä järjestemä.
CCE	Common configuration enumeration. Ylläpitää tunnisteita järjestelmien virheellisistä ja mahdollisesti haavoittuvaisista asetuksista.
CERT	Computer emergency readiness Team. Ylläpitää haavoittuvuustietokantaa, jonka haavoittuvuustiedot ovat pääasiassa yksityisen koordinoinnin ja julkistamisen tuloksia.
CVE	Common vulnerabilities and exposures. Lista yleisesti tunnetuista haavoittuvuuksista ja standardoitu tapa merkitä niitä.
CVSS	Common vulnerability scoring system. Standardoitu tapa ilmoittaa haavoittuvuuden ominaisuudet.
CWE	Common weakness enumeration. Yhteisön ylläpitämä lista yleisistä sovellushaavoittuvuuksista.
DAST	Dynamic application security testing. Käynnissä olevan sovelluksen tietoturvatestaamista mallintaen oikeita hyökkäyksiä.
DNS	Domain name system. Verkkoon kytkeytyneiden laitteiden nimeämisjärjestelmä.
HTTP	Hypertext transfer protocol. Selainten käyttämä tiedonsiirtoprotokolla.
IP-osoite	Internet protocol -osoite. Numeroitu osoite jokaiselle verkossa olevalle laitteelle.
JSON	JavaScript object notation. Kevyt merkintätapa tiedonvaihtoon.
MAC	Media access control. Verkossa toimivan laitteen valmistajan määräämä tunniste.

NIST	National institute of standards and technology. Yhdysvaltalainen teknologian standardeja ylläpitävä insitituutio.
NVD	National vulnerability database. Yhdysvaltojen ylläpitämä haavoittuvuustietokanta.
NVT	Network vulnerability test. OpenVAS-tietoturvaskannerin verkon haavoittuvuuksia analysoiva testi.
ORM	Object-relational mapping. Oliomallin mukaisen esityksen kuvaus relaatiomallin mukaiseksi esitykseksi.
OWASP	Open Web Application Security Project. Maailmanlaajuinen ja voittoa tavoittelematon sovellustietoturvan edistämiseen keskittynyt järjestö.
PCI	Payment card industry. Sille järjestetty neuvosto pitää huolta maksukorttijärjestelmille asetetuista tietoturvastandardeista.
REST	Representational State Transfer. Verkkopalveluiden arkkitehtuurimalli, jossa tieto välitetään JSON-muodossa.
SAST	Static application security testing. Tarkoittaa sovelluksen lähdekoodin tietoturvatestaamista.
URL	Uniform resource locator. Viittaa verkossa olevan resurssin sijaintiin.
WASC	Web application security consortium. Joukko asiantuntijoita, jotka luovat tietoturvasuosituksia verkkosovelluksille.
XML	Extensible markup language. Merkintäkieli tiedon tallentamiseksi ja siirtämiseksi.
ZAP	Zed attack proxy. OWASPin kehittämä ilmainen DAST-työkalu.

1 Johdanto

Insinöörityö tehdään konsultointiyritys Accenturelle. Se tarjoaa asiakkailleen muun muassa tietoturvaan liittyviä palveluita, joista yksi on sovellustietoturvasta huolehtiminen. Se tarkoittaa tietoturvan näkökulman tuomista sovelluskehityksen eri vaiheisiin.

Tietoturvalliseen sovelluskehitykseen kuuluu sovelluksen turvallisuuden testaaminen säännöllisesti, jotta mahdolliset haavoittuvuudet löydetään ja korjataan mahdollisimman aikaisessa vaiheessa. Testaaminen tehdään tietoturvaskannereilla, eli ohjelmilla, jotka selvittävät kohdejärjestelmän, kuten ohjelman tai palvelun alttiuden haavoittuvuuksille [1]. DAST-skannereilla tämä tehdään tarkkailemalla kohteen käyttäytymistä ja vastausta tietoturvatestin aikaiseen hyökkäykseen, ja SAST-työkalut taas tekevät johtopäätökset lukemalla kohteen lähdekoodia, ja etsimällä siitä merkkejä tunnetuista haavoittuvuuksista [2].

Tämä työ on osa Accenturen Django-ohjelmistorajapinnalla kehitettyä projektia, jossa käsitellään eri tietoturvaskannereilta tulleita raportteja samassa järjestelmässä. Skannereiden raportit eivät käytä standardoitua muotoa tai asteikkoa. Niiden erilaisuus johtuu myös skannerien eri käyttötarkoituksista, mittareista ja aineistoista. Rakenteellisiin eroihin vaikuttavat lisäksi työkalun kehittäjän omat päätökset. Tietoturvatestin lopuksi skannerit laativat raportin sen tuloksista ja tarjoavat sen JSON- tai XML-muodossa. Ne joko tallennetaan suoraan kovalevylle tai skannerin verkkorajapinnan kautta. Työstä rajattiin kokonaan pois raporttien hakeminen työkaluista, ja raportit löytyvät valmiiksi niille määrätystä kansiosta.

Itse raportti koostuu pääosin havaintoihin liittyvistä tiedoista (havainto-termi selitetään luvussa 3). Sisällöllisistä eroista johtuen ne täytyy uudelleenmuotoilla. Uutta rakennetta suunnitellessa huomioidaan, mitkä raporttien kentät ovat tärkeitä ja kuinka työkalukohtaisia ne ovat. Sen avulla täytyy myös voida suodattaa ja havainnollistaa eri lähteistä tulevia tuloksia. Raporttien uudelleenmuotoilu tullaan toteuttamaan niille räätälöidyillä komponenteilla, jotka käsittelevät niiden sisällön ja muotoilevat sen vastaamaan yhteistä rakennetta.

Projektin aikana pyritään selvittämään

- millaisia eri tietoturvaskannerit ovat
- millaisia raportteja ne tuottavat
- miten yhteensopivia raportit ovat keskenään
- millainen rakenne olisi paras ilmaisemaan raporttien tietoja
- mikä on tehokkain tapa raporttien käsittelyyn ja tallentamiseen.

Vaatimuksena yhteiselle tietorakenteelle on yhteensopivuus uusien palveluun lisättävien tietoturvaskannerien kanssa, eli uuden skannerin lisääminen projektiin tulisi tuottaa mahdollisimman vähän muutoksia. Rakenteen tulee myös soveltua tulosten käyttämiseen raportoinnissa, joten osaa kentistä tulee voida käyttää tulosten suodattamisessa ja kaavioiden piirtämisessä. Lisäksi havaintojen hallinnointitietoja pitää sisällyttää rakenteeseen mukaan, jotta niille voidaan muodostaa selvä elinkaari järjestelmässä. Jokaisesta havainnosta täytyy selvitä muun muassa sen tila, ja kenen vastuulla sen käsittely on.

Tietoturvaskannerit palauttavat välillä satoja havaintoja sisältäviä raportteja, jotka voivat paisua tästä syystä kymmeniin tuhansiin riveihin. Toteutuksessa pitää siis huomioida myös suurien aineistomäärien käsittely tehokkaasti. Koska työ toteutetaan osana olemassa olevaa projektia, ratkaisu tulee kehittää samalla alustalla ja yhteensopivasti.

Insinöörityö rakentuu pääpiirteittäin viidestä osasta. Ensimmäinen osa (luvut 2 ja 3) esittelee aiheen ja käsitteet, ja toinen osa (luku 4) tietoturvaskannerit yksi kerrallaan, joista kerrotaan esimerkiksi käyttötarkoitus ja ominaisuudet. Kolmannessa osassa (luku 5) käydään kaikkien edellisessä luvussa esiteltyjen työkalujen tuottamat raportit läpi ja siinä keskitytään niiden sisältöön ja rakenteeseen. Neljännessä osassa (luku 6) pohditaan, mikä olisi paras muoto, johon raporttien sisällön saisi yhdistettyä. Viimeisessä osassa (luku 7) esitellään eri tapoja käsitellä raportteja. Tämä luku jakaantuu kahteen osaan kokeiltujen toteutustapojen mukaisesti. Luvun loppu koostuu eri toteutusten suorituskykyjen vertaamisesta, ja tarkoituksena on päättää, kumpi esitellyistä toteutustavoista on tehokkaampi.

Metropolia

2 Taustatutkimus

Aiheeseen liittyviä tieteellisiä tutkimuksia ja artikkeleita etsiessä selvisi, ettei JSON- ja XML-tiedostojen jäsentämisessä luotavien objektien käsittelyä ole tutkittu kovin paljon. Aiheeseen liittyviä olennaisia kysymyksiä olisivat esimerkiksi, millä tekniikoilla objekteja on järkevää selata ja liittyykö tähän Pythonin kohdalla erityisiä huomioita. Suurin osa löytyneistä artikkeleista ja kirjoista liittyvät joko luonnollisen kielen tai tekstiformaatin, kuten XML, koneelliseen lukemiseen sekä näiden toimenpiteiden tehokkuuden vertailuun. Tämän työn ongelman kannalta ne ovat epärelevantteja, sillä työ liittyy enemmän tiedoston sisällön koneellisen lukemisen jälkeiseen jatkojalostamiseen, jossa sisältö muutetaan uuteen muotoon.

Aiheesta löytyy kuitenkin paljon keskustelua ja dokumentaatiota, ja esimerkiksi tekniikan johtaja Maxim Mamaev kirjoitti vuonna 2018 kattavan artikkelin silmukoiden toiminnasta Pythonissa [3]. Se ei kuitenkaan suoraan ota kantaa JSON- tai XML-rakenteiden käsittelyyn. Lisäksi on olemassa tätä työtä vastaava avoimen lähdekoodin projekti DefectDojo sekä kaupallinen työkalu CodeDx.

DefectDojo on OWASPin kehittämä havaintojenhallintaympäristö, jonka tarkoituksena on virtaviivaistaa havaintojen käsittelyä eri projektien sekä testikierrosten välillä [4; 5]. Siinä on 26 komponenttia erimuotoisten raporttien uudelleenmuotoiluun [6]. Muotoilu tapahtuu DefectDojon määrittelemän laajan havaintorakenteen mukaan [7], johon palataan luvussa kuusi. Se käyttää samaa ohjelmistokehystä kuin tässä insinöörityössä, eli Pythonin Djangoa. Tämän raportin aikana insinöörityöhön kehitettyjä ratkaisuja verrataan välillä DefectDojoon.

CodeDx pyrkii samaan kuin DefectDojo. Se kokoaa tietoturvatyökalujen tulokset yhteen hallinnointinäkymään. Havainnoista on myös mahdollista nähdä tilastoja, kuten kauanko aikaa havaintojen ratkaisemiseen menee keskimäärin ja kuinka aktiivisesti analyyseja suoritetaan [8]. CodeDx hinnoitellaan asiakaskohtaisesti.

3 Insinöörityön taustatietoja

Tämä luku rakentuu insinöörityön ohjelmistokehyksen sekä keskeisen havainto-termin määrittelystä. Ensimmäisenä esitellään Django-ympäristö, johon työ rakennetaan, minkä jälkeen selitetään mikä havainto on ja miksi termi tässä työssä keskeisessä osassa.

3.1 Django

Django on Python-kieleen perustuva avoimen lähdekoodin ohjelmistokehys, jonka suosio perustuu nopeaan ja vaivattomaan käyttöönottoon. Se tarjoaa monia verkkosovellukselle tärkeitä ominaisuuksia valmiina toteutuksina, joita ovat esimerkiksi käyttäjän- ja istunnonhallinta, lomakkeiden luonti sekä ORM [9; 10]. Tunnetuimpiin Djangoa käyttäviin verkkosivuihin lukeutuu muun muassa kuvienjakopalvelu Instagram [11] sekä versionhallintapalvelu Bitbucket [12].

Tämä työ on toteutettu erilliselle Django-projektille, jotta se olisi helppo yhdistää lopussa pääprojektiin. Ainoa käytetty Djangon ominaisuus on ORM, jonka avulla viedään havainnot ja siihen liittyvät tiedot tietokantaan. Djangon ORM toimii luokille, jotka on luotu sen models.py-tiedostoon. Näiden luokkien kentille voi määrittää Djangon model-kirjastoa käyttäen kenttätyypin, pää- ja viiteavaimet, rajoituksia ja oletusarvon, kuten esimerkkikoodissa yksi näkyy. Lisäksi koodin ”finding”-termi tarkoittaa havaintoa. Django luo luokkien perusteella tietokantataulun, ja sen kentät vastaavat taulun sarakkeita. [13, s. 84.]

```
class Link(models.Model):
    link = models.CharField(max_length=500,primary_key=True,unique=True)
    findings = models.ManyToManyField(Finding,related_name="links",on_delete=models.CASCADE)
```

Esimerkkikoodi 1. Link-luokka vastaa tietokannan Link-taulua, ja se periytyy Djangon Model-luokasta.

Koska jotkut koodinosat saattavat olla kiinnostuneita Djangon suorittamista tietokantatapahtumista, se tiedottaa niistä signaaleilla. Niitä lähetetään esimerkiksi ennen ja jälkeen tallentamisen sekä poistamisen, ja ne voivat käynnistää tiettyjä signaaleja kuuntelevia funktioita [13, s. 534–537]. Tässä työssä luotiin esimerkiksi pre_save-signaaliin reagoiva

funktio, joka suoritetaan aina ennen Finding-olion tallentamista. Jotta se voisi vastaanottaa signaalin, funktio täytyy ensin varustaa receiver-ilmoituksella, jossa määritellään, minkä luokan edustajan lähettämään signaaliin reagoidaan. Lisäksi funktio saa käyttöönsä olion, josta signaali on peräisin, instance-parametrissaan. Seuraavassa esimerkkikoodissa näkyy kyseisen funktion määritys.

```
@receiver(pre_save, sender=Finding)
def setPredefinedFields(sender, instance, *args, **kwargs):
```

Esimerkkikoodi 2. Pre_save-signaalin vastaanottava funktio.

Signaalia käytetään projektissa virheentarkistukseen, mihin palataan luvussa 6.

3.2 Havainto ja haavoittuvuus

Havainnolla (englanniksi ”finding”) tarkoitetaan tietoturvaskannerin ilmoittamaa tietoturvauhkaa, joka saattaa olla työkalusta riippuen haavoittuvuus, “väärä hälytys” tai bugi. Tätä termiä käytetään, ennen kuin tiedetään, mitä tietoturvatestin tuloksena on oikeasti löytynyt, eikä havaintoja haluta suoraan merkitä haavoittuvuuksiksi ennen tarkempaa tarkastelua. Skannerit ilmoittavat havainnoista, kun testissä käytetty sääntö palauttaa positiivisen tuloksen. Säännöt ovat määrityksiä esimerkiksi viallisesta ohjelmistokoodista tai asetuksesta, jonka löytyessä tunnistetaan haavoittuvuus. [14; 15.]

Haavoittuvuus taas on määritelty OWASPin toimesta tietoturva-aukoksi sovelluksessa, joka voi olla tyypiltään eri osapuolille harmia-aiheuttavan hyökkäyksen mahdollistava suunnitteluvirhe tai bugi [16]. Se täsmentää osapuolet sovelluksen omistajiksi, käyttäjiksi ja muiksi siitä riippuvaisiksi osapuoliksi ja hyökkäyksen haavoittuvuuden hyväksikäyttämiseksi [17]. Vaikka erilaisia haavoittuvuuksia on paljon, niistä yleisimmät on lueteltu OWASP TOP 10 -projektissa. Sen tekemisessä on mukana useita tietoturva-alan asiantuntijoita, ja se edustaa laajaa yhteisymmärrystä kymmenestä kriittisimmistä verkkosovelluksiin kohdistuvista haavoittuvuuksista. Projektin tarkoituksena on muuttaa ohjelmointikäytännöt tietoturvakeskeisimmiksi [18]. Vuonna 2017 julkaistussa listassa olivat seuraavat haavoittuvuudet [19]:

1. Injection. Epäluotettavasta lähteestä tulevaa aineistoa ei käsitellä oikein, mikä voi johtaa vahingollisen koodin suoritukseen ja tietovuotoihin.
2. Broken Authentication. Väärin toteutettu istunnonhallinta ja tunnistautuminen voivat johtaa käyttäjän identiteetin kaappaamiseen väliaikaisesti tai pysyvästi.
3. Sensitive Data Exposure. Verkkosovellus tai -rajapinta, joka ei tarjoa lisäsuojaa arkaluontointoisille tiedoille. Ne vaarantavat käyttäjänsä muun muassa luottokorttipetoksille ja identiteettivarkauksille, ja lisäsuojaa taas voi tuoda esimerkiksi tiedon salaaminen sekä sen välittämisen hienovarainen suunnittelu.
4. XML External Entities (XXE). Heikosti määritetty XML-käsittelijä käy läpi viitteet vieraisiin lähteisiin, mikä altistaa esimerkiksi palvelunestohyökkäyksille sekä sisäisten tiedostojen vuotamiselle.
5. Broken Access Control. Tunnistautuneiden käyttäjien oikeudet ja rajoitteet ovat määritetty huonosti, minkä takia hyökkääjä voi päästä käsiksi vääriin tietoihin ja toimintoihin.
6. Security Misconfiguration. Puutteelliset tietoturva-asetukset voivat tarkoittaa esimerkiksi turvattomia oletusasetuksia, puutteellisia asetuksia tai liian tarkkoja virheviestejä. Myös puutteelliset ohjelmistopäivitykset lukeutuvat tähän.
7. Cross-Site Scripting (XSS). Epäluotettavan datan sisällyttäminen verkkosivulla, ilman oikeaa käsittelyä, mahdollistaa käyttäjän istunnon kaappaamisen sekä sen ohjaamisen haitallisille sivuille.
8. Insecure Deserialization. Sarjoituksen poistaminen turvattomasti voi johtaa vieraan koodin tahattomaan suoritukseen ja altistavat hyökkäyksille, kuten injektio ja valtuuksien korotus.
9. Using Components with Known Vulnerabilities. Sovelluksen käyttämät riippuvuudet suoritetaan samoilla valtuuksilla kuin sovellus, joten sellaisesta löytyvä haavoittuvuus voi heikentää tietoturvaa. Seurauksena voi olla esimerkiksi palvelimen kaappaaminen.
10. Insufficient Logging & Monitoring. Jos sovellusta ei tarkkailla ja lokiteta tarpeeksi hyökkäykseen reagointi on hidasta, ja se sallii hyökkääjän tehdä enemmän ja jatkuvaa vahinkoa.

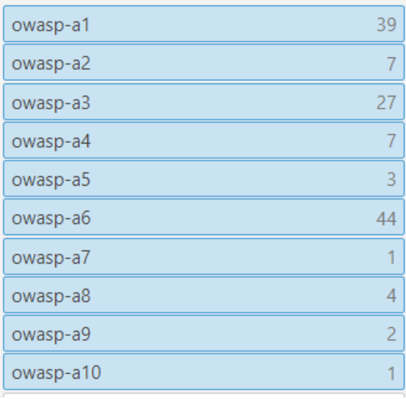
Taulukossa 1 on vielä listattu tämän työn kohteena olleet työkalut ja niiden soveltuvuus näiden haavoittuvuuksien tunnistamiseen. Työkalut esitellään taulukon jälkeen seuraavassa luvussa.

Taulukko 1. Jokainen rivi edustaa yhtä OWASP TOP 10 -haavoittuvuutta, sarakkeet taas ovat tässä työssä käytettyjä tietoturvatyökaluja.

Haavoittuvuus	**SonarQube**	**Retire.js**	**Dependency-Check**	**ZAP**	**OpenVAS**	**Retina**
Injection	X			X		
Broken Authentication	X			X	X	X

Sensitive Data exposure	X			X	X	X
XML external entities	X			X		
Broken access control	X			X	X	X
Security misconfiguration	X			X	X	X
XSS	X			X		
Insecure deserialization	X			X		
Using components with known vulnerabilities	X	X	X	X		
Insufficient logging and monitoring	X			X		

SonarQube ja Zed Attack Proxy tunnistavat kaikista OWASP TOP 10 -haavoittuvuuksista joitakin tapauksia, mutta eivät välttämättä kaikissa tilanteissa [20, 21]. Esimerkiksi SonarQubella on osalle haavoittuvuuksia vain yksittäisiä sääntöjä, eli suurin osa sen tukemista kielistä edes välttämättä tunnista kaikentyyppisiä haavoittuvuuksista. Kuvasta 1 näkee tarkemmin, kuinka monta sääntöä jokaiselle TOP 10 -haavoittuvuudelle siitä löytyy. ZAP taas antaa lähinnä suuntaviivoja erilaisten haavoittuvuuksien testaamiseen listaamalla, millä sen työkaluilla niitä voisi mahdollisesti löytää.



Kuva 1. Vasemman puolen teksti tarkoittaa SonarQuben merkintää kustakin OWASP TOP 10 -haavoittuvuudesta, ja numerot kertovat, montako sääntöä on olemassa jokaista haavoittuvuutta kohden.

Retire.js ja Dependency-Check kykenevät ainoastaan tunnistamaan haavoittuvaisia riippuvuuksia, mikä vastaa suoraan taulukon yhdeksännettä kohtaa. Raporttien perusteella Retina ja OpenVAS ovat puolestaan verkkoihin keskittyneitä haavoittuvuusskannereita, ja tunnistavat tunnistautumiseen, käyttöoikeuksiin sekä asetuksiin liittyviä ongelmia.

4 Tietoturvaskannerit

Työssä käsitellään kuutta eri tietoturvaskanneria, joista puolet ovat SAST- ja puolet DAST-skannereita. Ensimmäisiin kuuluvat OWASPin kehittämät SonarQube sekä Dependency-Check ja Node.js moduuli Retire.js. Jälkimmäisiin taas lukeutuvat OpenVAS, OWASPin ZAP ja Retina. Näiden valinta perustui ilmaisuuteen, pätevyyteen, automaatiokelpoisuuteen sekä tuettuihin ohjelmointikieliin.

Ilmaisten työkalujen kohdalla ei voi olla kovinkaan nirso, sillä tarjonta ei ole laajaa, ja useat kehittäjät ovat alkaneet siirtää tukea ilmaisversiosta maksullisiin. Esimerkiksi SAST-työkalu Nessuksen kohdalla vuoden 2018 alussa tullut päivitys poisti mahdollisuuden automatisoida tietoturvaskannaus sen rajapinnan kautta, mikä painostaa käyttämään saman yrityksen omistamaa maksullista Tenable.io-pilvipalvelua, josta tämä ominaisuus vielä löytyy [22]. Lisäksi tässä työssä käytetyn OpenVASin haavoittuvuustietokannan päivittäminen päätettiin lopettaa vuoden 2017 jälkeen, koska kehittäjien mielestä ilmaisversion ei tarvitse tuottaa yhtä tarkkoja tuloksia kuin maksullisen [23]. Tämä pakottaa sen vaihtamiseen toiseen työkaluun pidemmällä aikavälillä.

Skannerin tarkoitus on löytää haavoittuvuuksia ja antaa niistä hyödyllistä tietoa. Pätevyydellä tarkoitetaan sen kykyä tehdä tätä, ja sitä mitattiin testiajoilla valmiiksi haavoittuvaisien ohjelmien kanssa. Niitä ovat esimerkiksi valmiiksi rikkinäinen verkkosovellus WebGoat [24] tai virtuaalikone Metasploitable [25]. Niiden avulla työkalun antamasta raportista löytyy paljon erityyppisiä havaintoja ja saa hyvän kuvan niiden määrästä, tyypeistä ja tiedoista.

Jotta skanneria voisi käyttää tietoturvatestien ajamiseen automaattisesti ja itsenäisesti, sen tulee toimia Python-koodista käsin käyttäjän hyväksi työkalun komentoriviä tai verkkorajapintaa. Skannauskohteiden lisääminen, testien aloittaminen ja raporttien hakeminen ovat tärkeimmät toiminnot, jotka pitää voida suorittaa käyttäen Pythonia, mikä rajaa ulos ne työkalut, joilla on pelkkä graafinen käyttöliittymä. Kun koko testiprosessia voi hallita koodista käsin, sen automatisointi on yksinkertaista.

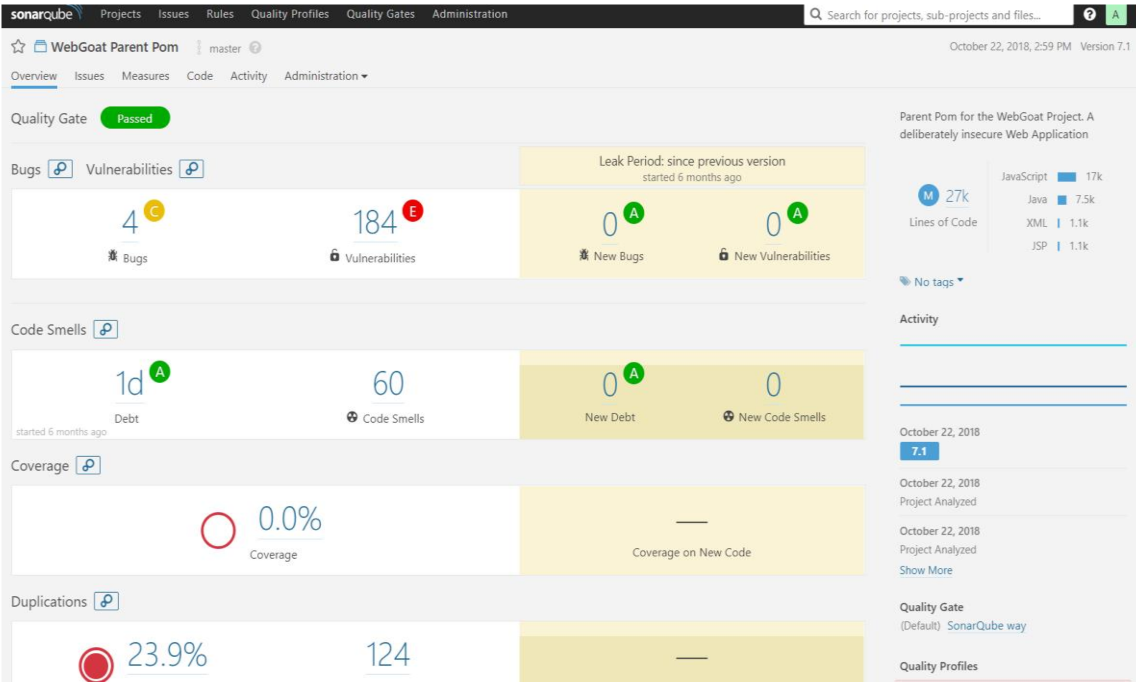
Työkalujen valintaan vaikuttavat myös niiden tukemat ohjelmointikielet ja teknologiat. Koska tämän työn tuotoksia tullaan todennäköisesti käyttämään Java-projekteissa, ovat

esimerkiksi staattiset skannerit painottuneita siihen. Myös kielen yleisyys vaikuttaa valitsemiseen. Esimerkiksi Javascript-skanneria voi hyödyntää jokaisessa projektissa, jossa rakennetaan ollenkaan verkkokomponentteja. Osa valituista skannereista on kohteen ohjelmointikielestä riippumattomia ja testaavat enemmän asetuksia ja toiminnallisia virheitä.

Jotta jokaisesta työkalusta saisi alustavan mielikuvan, niiden esittely alkaa käymällä sen käyttöliittymä pääpiirteittäin läpi kuvan avulla. Sen jälkeen kerrotaan ominaisuuksista ja erottavista tekijöistä, ja seuraavassa luvussa tutkitaan niiden raportteja tarkemmin.

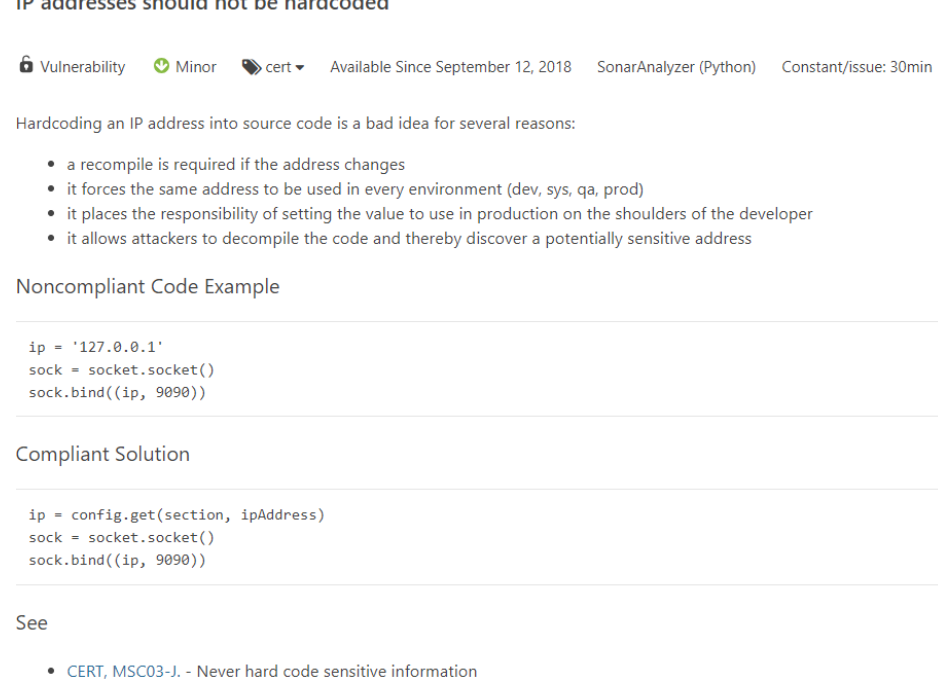
4.1 SonarQube

SonarQube on Java-pohjainen vuonna 2007 alkunsa saanut avoimen lähdekoodin tietoturvaskanneri [26]. Kuvassa kaksi on esimerkki sen käyttöliittymästä. Kuvassa näkyvällä projektisivulla näytetään havaintojen määrä, laatu ja trendi. Muuten käyttöliittymä soveltuu käyttäjien ja havaintojen hallintaan. Itse skannaus on kuitenkin suoritettava komentoriviltä.



Kuva 2. SonarQuben käyttöliittymä on selainpohjainen.

Ilmaisen version lisäksi SonarQubesta löytyy maksullisia vaihtoehtoja. Niissä tulee mukana muun muassa tuki useammille ohjelmointikielille. Tosin näistä yleisimmät kuuluvat ilmaisversioonkin [27]. SonarQube toimii käymällä lähdekoodin läpi etsien tiettyjä piirteitä sisäisten sääntöjen perusteella. Ne voivat kuvata joko bugeja, huonoja koodauskäytäntöjä tai haavoittuvuksia. Seuraava kuva on esimerkki SonarQuben tietoturvasäännöstä.



Kuva 3. SonarQuben sääntösivut antavat kuvauksen ja esimerkin säännön löytämästä haavoittuvuudesta sekä sen korjaamisesta. Se tapahtui tässä kuvassa kovakoodatun IP-osoitteen siirtämisellä pois ohjelman lähdekoodista.

SonarQube tukee yli 25 eri kieltä, ja sääntöjen määrä vaihtelee kielen mukaan [28]. Esimerkiksi Pythonille on pelkästään yksi haavoittuvuussääntö [29; 30], eli sen lähdekoodianalyysi on keskittynyt löytämään lähinnä bugeja ja huonoja koodauskäytäntöjä. Javalle haavoittuvaisen koodin tunnistamiseen on taas luotu 49 sääntöä, joilla tietoturvaan liittyviä virheitä tunnistetaan huomattavasti enemmän. Lisäksi SonarQubeen on mahdollista asentaa liitännäisiä, kuten sääntöjä uusille ja olemassa oleville kielille. Tässä työssä käytetään FindBugs-liitännäistä, joka integroi SpotBugs-tietoturvaskannerin SonarQubeen. Käyttöliittymässä tämä näkyy 124 lisäsääntönä SonarQuben Java-skanneissa, eli havaintoja pitäisi löytyä paljon enemmän. Muita liitännäisiä ovat esimerkiksi kielten lokalisaatiot ja palveluiden käyttäjäntodennuskomponentit, jotka sallivat tunnistautumisen esimerkiksi Google-tilillä.

Tietoturvaa testatessa ei välttämättä olla kiinnostuneita muista kuin haavoittuvuuksiin liittyvistä havainnoista, minkä takia SonarQube tarjoaa mahdollisuuden luoda laatuprofiileja. Ne ovat testeissä käytettäviä sääntöyhdistelmiä, joiden avulla turhista havainnoista päästään eroon ja tärkeitä havaintoja löytyy enemmän. SonarQuben hyödyllisyys tietoturvatestauksessa on paljolti riippuvainen hyvistä määrityksistä. Kuvassa 4 on esimerkki tietoturvatestauksessa käytettävästä Java-ohjelmointikielen laatuprofiilista.

Rules	Active	Inactive
Total	173	1.2k
Bugs	0	561
Vulnerabilities	173	0
Code Smells	0	609

Activate More

Kuva 4. Tietoturvaskannissa laatuprofiiliin otetaan mukaan ainoastaan haavoittuvuussääntöjä, sillä muut bugeihin ja koodin suunnitteluvirheisiin liittyvät säännöt aiheuttavat liikaa tarpeettomia havaintoja.

Lisäksi on mahdollista asettaa laatuportteja, jotka määrittävät, milloin skannattun projektin koodinlaatu on hyväksyttävä. Mittauskohteita ovat projektin koodikattavuus ja tietoturvallisuus, kaksoiskoodin määrä, koodin ylläpidettävyys ja luotettavuus. Laatuportit ovat siis SonarQuben sisäinen tapa määrittää projektin koodin kelpoisuus.

SonarQubesta löytyy hallintapaneeli jokaiselle skannatulle projektille, jonka kautta statistiikkaa, lähdekoodia ja havaintoja selataan. Se kertoo havaintojen esiintymismääristä sekä antaa mahdollisuuden merkitä havaintoja virheellisiksi ja korjatuksi. SonarQube tarjoaa visuaalisen käyttöliittymän lisäksi REST-rajapinnan havaintojen hakemiseksi. Tässä työssä käsiteltävä SonarQuben raportti on rajapinnan palauttama JSON-tiedosto testiprojektin havainnoista.

4.2 Riippuvuuksia testaavat SAST-skannerit

Sekä Dependency-Check että Retire.js vertaavat projektin lähdekoodin riippuvuuksia haavoittuvaisten riippuvuuksien listaa vasten ilmoittaen, jos tulos on positiivinen. Työkalut osaavat myös kertoa riippuvuudesta löytyvät riskit ja mihin versioon riippuvuus kannattaisi päivittää. Kuten aiemmassa luvussa tuli ilmi, haavoittuvaisten komponenttien käyttäminen on osa OWASPin kymmenen kriittisintä tietoturvariskiä -listaa, joten riippuvuusskannereille on tarvetta.

Retire.js tarkistaa Javascript-riippuvuuksien turvallisuuden luetteloaan vasten ja on kevyt komentoriviohjelma. Seuraavassa kuvassa on sen apusivu, jolla kerrotaan retire-komentoon liittyvistä asetuksista. Retire.js kattaa noin sata Javascript-kirjastoa [31], ja tulokset palautetaan JSON-muodossa käyttäen "--outputformat json" -asetusta.

```
 Usage: retire [options]

 Options:

   -h, --help                output usage information
   -V, --version             output the version number

   -p, --package             limit node scan to packages where parent is mentioned in package.json (ignore node_modules)
   -n, --node                Run node dependency scan only
   -j, --js                  Run scan of JavaScript files only
   -v, --verbose             Show identified files (by default only vulnerable files are shown)
   -x, --dropexternal        Don't include project provided vulnerability repository
   -c, --nocache             Don't use local cache

   --jspath <path>           Folder to scan for javascript files
   --nodepath <path>         Folder to scan for node files
   --path <path>             Folder to scan for both
   --jsrepo <path|url>       Local or internal version of repo
   --noderepo <path|url>     Local or internal version of repo
   --cachedir <path>         Path to use for local cache instead of /tmp/.retire-cache
   --proxy <url>             Proxy url (http://some.sever:8080)
   --outputformat <format>   Valid formats: text, json
   --outputpath <path>       File to which output should be written
   --ignore <paths>          Comma delimited list of paths to ignore
   --ignorefile <path>       Custom ignore file, defaults to .retireignore / .retireignore.json
   --severity <level>        Specify the bug severity level from which the process fails. Allowed levels none, low, medi
um, high, critical. Default: none
   --exitwith <code>         Custom exit code (default: 13) when vulnerabilities are found
   --includemeta             Include version and scan time in JSON result
```

Kuva 5. Asetusten avulla voi määrittää muun muassa, mitä sijaintia ja minkälaisia tiedostoja Retire.js skannaa, missä muodossa ja mihin raportti viedään sekä tarvitseeko tiettyjä tiedostoja jättää huomioimatta.

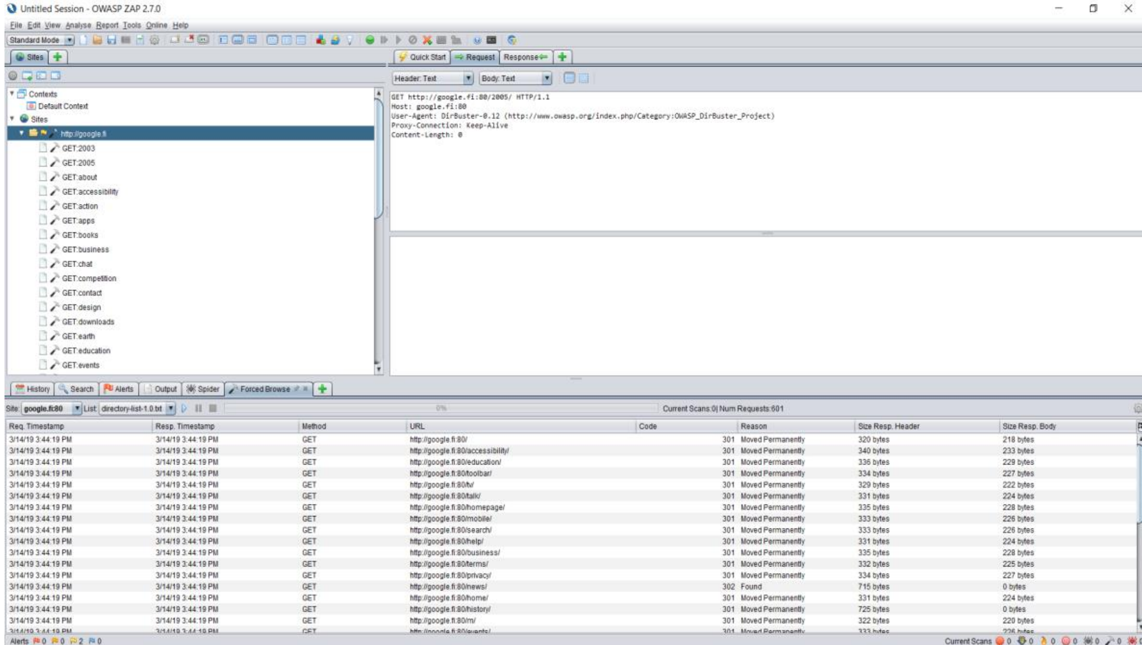
OWASPin kehittämä Dependency-Check on taas keskittynyt Java- ja .NET-koodin skannaamiseen ja saa tietonsa haavoittuvaisista riippuvuuksista NISTin ylläpitämästä NVD-tietokannasta. Se on huomattavan laaja, ja yhden kuukauden aikana havaintojen määrä

saattaa lisääntyä yli tuhannella haavoittuvuudella [32]. Retire.js:n tavoin se on komentoriviohjelma, ja myös niiden asetukset ovat suurelta osin samanlaiset.

Dependency-Checkin toimintalogiikka on seuraava: kohdetiedostot analysoidaan ja niistä kerätään riippuvuuteen liittyviä tietoja, joita kutsutaan todisteiksi. Todisteita verrataan eri haavoittuvuuksiin liittyviin riippuvuuksiin, ja yhteenosuman tapauksessa todisteeseen yhdistetään haavoittuvuus [33]. Dependency-Check palauttaa helppolukuisen HTML-raportin sekä XML-raportin.

4.3 Zed Attack Proxy

OWASPin kehittämä ZAP on ilmainen ja avoimen lähdekoodin työkalu verkkosovellusten tietoturvan dynaamiseen testaamiseen. Se on työpöytäsovellus, joka soveltuu useiden kohteiden ja testien hallinnoimiseen samanaikaisesti [kuva 6].



Kuva 6. ZAP-käyttöliittymän vasemmassa reunassa listataan kaikki senhetkiset tunnetut kohteet, joihin sen työkaluja voi käyttää. Tunnettuja kohteita voi lisätä joko itse, tai niitä voi etsiä automaattisesti. Oikeassa reunassa voi tarkastella valittua HTTP-pyyntöä ja sen vastausta, ja alapalkki koostuu eri testien tuloksista.

ZAPin ominaisuuksia ovat automaattiset haavoittuvuusskannit sekä mahdollisuus manuaaliseen penetraatiotestaukseen [34], joka tarkoittaa verkkoon tai tietojärjestelmään kohdistuvan hyökkäyksen simuloimista [35]. ZAPin automaattinen skannaus voidaan jakaa passiiviseen ja aktiiviseen skannaukseen, joilla voi löytää yksinkertaisia haavoittuvuuksia. Monimutkaisemmat tapaukset, kuten sovelluksen käyttäytymislogiikkaan perustuvat haavoittuvuudet, jäävät niiltä kuitenkin huomaamatta ja vaativat manuaalista testausta [36]. Tässä työssä keskitytään ainoastaan automaattisilla työkaluilla saataviin tuloksiin.

Aktiivinen skannaus käyttää tunnettuja hyökkäyksiä kohdetta vastaan, ja sen käyttämiseen tarvitaan lupa kohdepalvelun tai -palvelimen omistajalta. Se lähettää kuhunkin hyökkäykseen liittyvää sisältöä kohteeseen ja kerää tulokset hyökkäyksen onnistumisesta [37]. Kuvassa 7 on esimerkki aktiivisen skannauksen tekemästä ”Directory traversal” -tyypin hyökkäyksestä.

Method	URL	Code
GET	http://google.fi:80/	301
GET	http://google.fi:80/accessibility/	301
GET	http://google.fi:80/education/	301
GET	http://google.fi:80/toolbar/	301
GET	http://google.fi:80/tv/	301
GET	http://google.fi:80/talk/	301
GET	http://google.fi:80/homepage/	301
GET	http://google.fi:80/mobile/	301
GET	http://google.fi:80/search/	301
GET	http://google.fi:80/help/	301
GET	http://google.fi:80/business/	301
GET	http://google.fi:80/terms/	301
GET	http://google.fi:80/privacy/	301
GET	http://google.fi:80/news/	302
GET	http://google.fi:80/home/	301
GET	http://google.fi:80/history/	301
GET	http://google.fi:80/m/	301
GET	http://google.fi:80/events/	301

Kuva 7. Aktiivisen skannauksen tekemä hyökkäys Googlelle. Directory traversal -hyökkäys pyrkii löytämään piilotettuja polkuja ja tiedostoja palvelimelta käyttäen valmista sanalistaa. Se koostuu yleisistä hakemistojen nimistä, jotka hyökkäyksessä lisätään URL-osoitteen loppuun.

Passiivinen skannaus taas ei ole hyökkäys, vaan se analysoi taustalla tietoliikennettä kohteen ja ZAPin välillä ja ilmoittaa havainnoista [38]. Passiivista skannausta varten ZAP on asetettava välityspalvelimeksi, jolloin kaikki verkkoliikenne kulkee sen kautta. Tämä sallii liikenteen automaattisen sekä manuaalisen tarkkailun. Kuvassa kahdeksan on esi-

merkki palvelimen lähettämästä HTTP-vastauksesta, joita passiivinen skannaus tarkkailee. SonarQuben tapaan ZAP perustuu Javaan ja sallii liitännäisten käytön vakio-ominaisuuksien ohella. Lisäksi raportin voi hakea ZAPin REST-rajapinnan kautta joko XML- tai JSON-muodossa.

```
HTTP/1.1 301 Moved Permanently
Location: https://www.google.fi/robots.txt
Content-Type: text/html; charset=UTF-8
X-Content-Type-Options: nosniff
Date: Fri, 08 Mar 2019 05:43:51 GMT
Expires: Sun, 07 Apr 2019 05:43:51 GMT
Server: sffe
Content-Length: 229
X-XSS-Protection: 1; mode=block
Cache-Control: public, max-age=2592000
Age: 546561
```

```
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="https://www.google.fi/robots.txt">here</A>.
</BODY></HTML>
```

Kuva 8. Passiivisessa skannauksessa käydään saatuja HTTP-vastauksia läpi etsien haavoittuvuuksia sen HTML-koodista ja -otsikoista.

ZAPin keskeisiä vahvuuksia on sen laaja tuki tietoturvatestien automatisoinnille. Sen rajapintaa voi käyttää viidellä eri ohjelmointikielellä, mukaan lukien Pythonilla, koodikirjastojen avulla. Myös koontityökaluille Maven ja Ant löytyy mahdollisuus ajaa ZAP-tehtäviä, jotka voivat aloittaa sen skannit koonnin yhteydessä automaattisesti. [39.]

4.4 OpenVAS

OpenVAS on Greenbonen omistama avoimen lähdekoodin haavoittuvuusskanneri, joka haarautui tietoturvatyökalu Nessuksesta vuonna 2005 tämän lähdekoodin muuttuessa

suljetuksi [39]. Siitä löytyy ilmaisen version lisäksi viisi maksullista yrityksille suunnattua versiota, joiden ominaisuudet vaihtelevat yrityksen koon ja IP-avaruuden mukaan [40].

OpenVAS toimii selaimella ja komentorivillä. Selainnäkymä muodostaa koosteen kaikkien skannausten tuloksista ja mahdollistaa uusien kohteiden, tehtävien ja skannausten lisäämisen [kuva 9]. Automatisointi tulee kuitenkin toteuttaa komentoriviltä käsin [kuva 10].

Kuva 9. OpenVASin graafisen käyttöliittymä toimii selaimella Greenbone Security Assistant -palvelun kautta.

OpenVAS toimii skannaamalla kohteen verkkoinfrastruktuurin haavoittuvuuksia ulkoapäin, toisin sanoen hyökkääjän näkökulmasta [41]. Kohde määritetään IP-osoitteella. OpenVAS analysoi kohteessa sijaitsevia palveluita, avoimia portteja ja asetuksia, eli mahdollisimman paljon asioita, jotka näkyvät kohteesta ulkoverkkoon. Analyysissä käytettyjä verkkohaavoittuvuustestejä (NVT) on yli 50 000. Lisäksi testejä tulee jatkuvasti lisää, mutta ainoastaan maksullisiin versioihin, koska vuoden 2017 lopussa ilmaisversion testikannan päivitys lopetettiin [23].

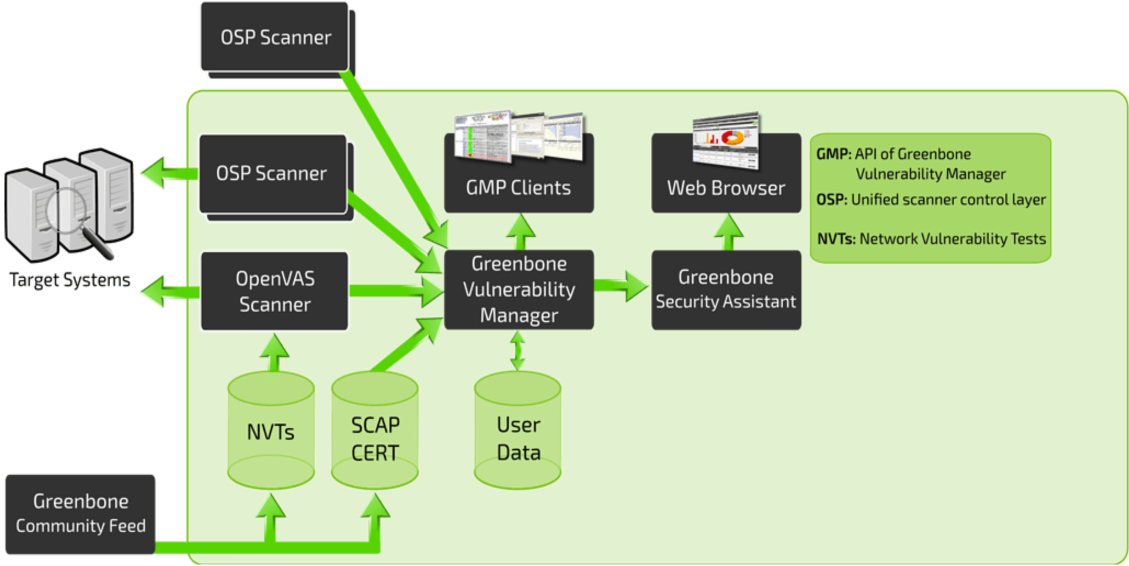
Rakenteeltaan OpenVAS jakaantuu kolmeen pääosaan: manageriin, skanneriin ja Greenbonen tietoturva-assistenttiin (GSA). Manageri mahdollistaa haavoittuvuuksien hallintaan liittyvät toimenpiteet, kuten käyttäjien ja tietokannan hallinta. Lisäksi se käyttää

sisäistä protokollaa skannerikomponentin käskyttämiseen ja mahdollistaa itsensä kanssa kommunikoinnin XML-pohjaisella “Openvas Management” -protokollalla (OMP). OMP muun muassa sallii OpenVASin käytön komentoriviltä ja täten sen automatisoinnin. XML-komennot voi kirjoittaa suoraan komentoriville, kuten kuvassa 10 tehdään, mutta selkeyden vuoksi suositeltavaa olisi kirjoittaa ne ensin tiedostoon, ja ajaa OMP-komento sitä käyttäen. [42; 43.]

```
otso@ubuntu:~$ omp --username admin --password admin --xml='<create_task>
> <name>First scan</name>
> <config id="8715c877-47a0-438d-98a3-27c7a6ab2196"/><target id="c39fdcb5-dc2b-47e9-94ff-f576c2cc4536"/>
> </create_task>'
```

Kuva 10. Tunnistearvot piti ensin hakea toisilla OMP-komennoilla, ja liittää vastauksista tähän komentoon. OpenVASin ohjaaminen XML:llä tuntuu vanhanaikaiselta, mutta se on toimiva tapa automatisoida sen toiminta.

GSA on OpenVAS:n käyttöliittymänäkymä, jonka kautta ohjelma toimii ilman komentorivin kanssa kamppailua. Sen avulla voi selata raportteja löytyneistä haavoittuvuuksista ja kohteista, aloittaa skanneja ja hallita käyttäjiä. Se siis tarjoaa vaihtoehtoisen tavan kommunikoida managerille konsolin sijaan. Seuraavana on vielä havainnollistava kuva koko OpenVAS-arkkitehtuurista.

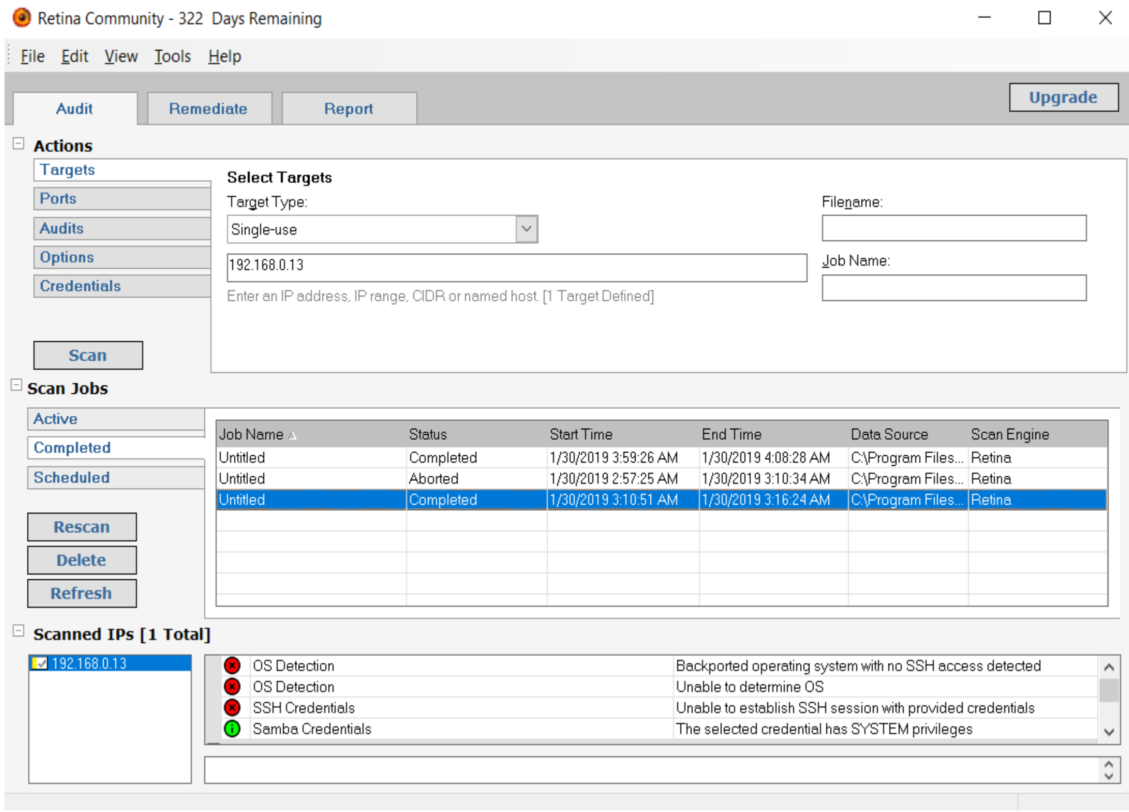


Kuva 11. Edellä selitetty rakenne, GMP tarkoittaa OMP:n uudempaa versiota, Greenbone Management -protokollaa, joka ajaa saman asian. [43]

OpenVAS raportti on XML-muotoinen, joskin ihmiselle sen lukeminen on hankalaa. Esimerkiksi tämän työn testiraportissa oli 27 riviä, jotka olivat jopa 150 000 merkkiä pitkiä. Uudelleenmuotoilun tuloksena sen rivimäärä kasvoi 37 000:een. Koneellisesti raportin lukeminen käy ongelmattomasti. Raportista kerrotaan lisää luvussa viisi.

4.5 Retina

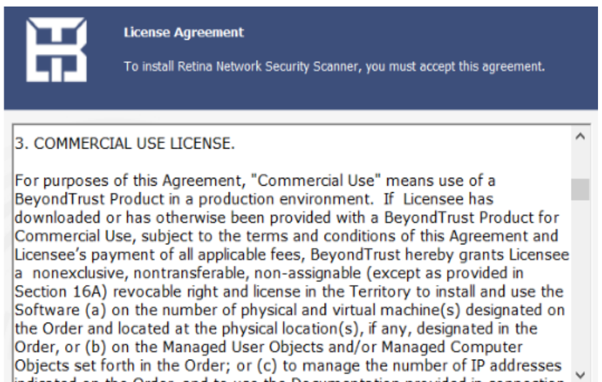
Vuonna 1998 BeyondTrustin luoma Retina-verkkoskannaustyökalu, joka on toiminnaltaan ja tarkoitukseltaan verrattavissa OpenVASiin. Sillä pyritään havaitsemaan kohdeverkon uhkia ja hallinnoimaan niitä OpenVASin GSA-tyyppisessä raportointinäkymässä. Retina on työpöytäsovellus [kuva 12], mutta se toimii osittain myös komentorivin kautta.



Kuva 12. Retinan skannausnäkymässä voi muun muassa lisätä uusia kohteita, aloittaa skannauksia sekä selata niiden tuloksia.

Retina on kotikäytössä ilmainen, mutta kaupallinen käyttö vaatii maksullisen lisenssin. BeyondTrustin verkkosivut ovat tässä mielessä sekavat, sillä mikään ei viittaa työkalun

maksullisuuteen, vaan pelkästään itse lisenssissä lukee kaupallisen käytön maksava [kuva 13]. Kaupallisen lisenssin kohdalla mainitaan ohjelman olevan maksullinen tuotantokäytössä, mikä onkin oletettavaa. Kuitenkaan tietoja eri lisensseistä tai Retinan ostosivua ei löydy helposti. [44; 45; 46.]



License Agreement

To install Retina Network Security Scanner, you must accept this agreement.

3. COMMERCIAL USE LICENSE.

For purposes of this Agreement, "Commercial Use" means use of a BeyondTrust Product in a production environment. If Licensee has downloaded or has otherwise been provided with a BeyondTrust Product for Commercial Use, subject to the terms and conditions of this Agreement and Licensee's payment of all applicable fees, BeyondTrust hereby grants Licensee a nonexclusive, nontransferable, non-assignable (except as provided in Section 16A) revocable right and license in the Territory to install and use the Software (a) on the number of physical and virtual machine(s) designated on the Order and located at the physical location(s), if any, designated in the Order, or (b) on the Managed User Objects and/or Managed Computer Objects set forth in the Order; or (c) to manage the number of IP addresses indicated on the Order, and to use the Documentation provided in connection

Kuva 13. Vasemmalla on esimerkki Retinan verkkosivuilla olevista ilmaisuuteen viittaavista teksteistä. Oikealla on Retinan lisenssi, jossa selitetään sen eri tyypit.

Tämän työkalun automatisointi osoittautui haastavaksi. Tietoturvaskannien aloitus onnistuu komentoriviltä, mutta uuden luominen ei ole mahdollista, ja se on luotava ensin manuaalisesti käyttöliittymän kautta [47]. Myöskään dokumentaatiota ei ole paljon verrattuna muihin työkaluihin, vaikka se on yli kaksikymmentä vuotta vanha.

Skannauskohteita ovat OpenVASin tapaan kohteen käyttöjärjestelmä, sen asetukset ja päällä olevat palvelut sekä portit. Retina käy läpi lisäksi IoT-laitteet tarkistaen, onko niissä oletustunnukset käytössä. Sen virheellisyysprosentti on alle yhden, mutta toisaalta Retinan testit ovat myös hyvin hitaita. Retinan käyttämän haavoittuvuustietokannan väitetään olevan laaja ja usein päivittyvä, muttei tälle löydy tarkkaa lähdettä. [48.]

5 Raportit

Edellisessä luvussa esiteltyt työkalut palauttavat joko XML- tai JSON-muotoisen raportin skannauskohteesta löytyneistä havainnoista. Raporttien sisältö vastaa usein graafisesta käyttöliittymästä löytyviä tietoja. Havaintoihin liittyy myös usein tunnistenumero, jolla se on listattu skannerin käyttämässä haavoittuvuustietokannassa. Tunnistenumeron ja mahdollisten linkkien avulla on mahdollista etsiä lisätietoja havainnosta. Suurimmat erot raporttien välillä tulevat niiden tavasta jäsennellä tietoa ja sisällön laajuudesta, mutta jokaisesta on mahdollista erottaa eri havaintoihin liittyvät tiedot toisistaan. Raportit luotiin ajamalla tietoturvaskannerit tarkoituksella haavoittuvaisia kohteita vasten, minkä ansiosta niihin tuli varmasti sisältöä. Lopuksi ne tallennettiin kansioon kiintolevylle.

5.1 Common vulnerability scoring system (CVSS)

Osa raporteista käyttää CVSS-mallia. Se on avoin haavoittuvuuksien luokittelua ja seurauksia määrittävä rajapinta, jonka tarkoituksena on yhdenmukaistaa niiden arviointi. Tällä hetkellä iso osa löydetyistä haavoittuvuuksista arvioidaan eri asteikoilla, mikä vaikeuttaa niiden keskinäistä hallinnoimista ja priorisointia.

CVSS arvioi haavoittuvuuksia perus-, väliaikais- ja ympäristöryhmissä. Perusryhmä kuvastaa haavoittuvuuden sisäistä luonnetta ja määrittää haavoittuvuuden perusominaisuudet. Väliaikaisryhmä taas kuvaa niitä piirteitä, jotka muuttuvat ajan myötä, ja ympäristöryhmä tuo esille käyttäjän ympäristön piirteet, jotka liittyvät haavoittuvuuteen. Tässä työssä olennaisinta on perusryhmän arviointiperusteet. Ne koostuvat vakavuusasteesta, joka on nollan ja kymmenen välillä ja heijastaa havainnon uhkaavuutta sekä hyökkäysvektorin tyypistä. Vektoreita ovat lokaali verkko, lähiverkko ja julkinen verkko, ja ne kuvaavat, mistä mahdollinen hyökkäys voi tulla. Lisäksi perusryhmässä arvioidaan uhkan hyväksikäyttämisen helppoutta, tunnistautumisen tarvetta ja onnistuneen hyökkäyksen seurauksia. [49.]

5.2 SonarQube

Raportiltaan SonarQube on selkeä. Se on JSON, joka koostuu sivu- ja havaintotiedoista sekä kohteista, säännöistä, käyttäjistä ja koosteesta.

SonarQuben rajapinta palauttaa tulokset sivuina, joilla on enimmäiskoko. Jos esimerkiksi havaintoja on 248, ja ensimmäinen rajapintakutsu palauttaa niistä vain 100, tiedetään, että seuraava kutsu täytyy tehdä toiselta sivulta alkaen. Tämän jälkeen haetaan vielä 48 havainnon kolmas sivu. Sivutiedot ilmaisevat raportissa siis, monennenko sivun tiedot on palautettu, mikä on enimmäiskoko ja paljonko havaintoja on yhteensä.

Jokainen SonarQuben raportti sisältää listan havainnoista. Niiden tietoihin taas kuuluvat vakavuusaste ja tila, sekä mistä ne löytyivät ja millä säännöllä. Sääntökentässä on viiteavain raportissa myöhemmin sijaitsevaan sääntölistaan, joka kertoo siitä lisätietoja. SonarQuben vakavuusasteikko on “info”, “minor”, “major”, “critical” ja “blocker” lueteltuna vähiten vakavasta vakavimpaan. Tilatiedot kertovat haavoittuvuuden tilan SonarQuben omassa hallinnointijärjestelmässä. Ne kuvaavat sen elinkaarta; tila asetetaan oletuksena avoimeksi, ja se muuttuu suljetuksi tai vahvistetuksi käyttäjien tehdessä havainnolle tarvittavat toimenpiteet. Muita tiloja on uudelleen avattu ja vahvistettu [50]. Havainnon paikkatietoina SonarQube raportoi projektin, tiedoston sekä rivit, joilla se ilmeni. Näiden avulla voi siis rajata ongelman lähteen rivin tarkkuudella oikeassa tiedostossa, ja erottaa ne projektien perusteella. Muita havaintotietoja ovat esimerkiksi haavoittuvuuden ohjelmointikieli, omistaja ja kuvaus.

Loput raportista ovat tarkennuksia esimerkiksi projektista, käyttäjistä ja säännöistä. Lisäksi raportista löytyy tilastotietoja, kuten montako minkäkin tyyppistä havaintoa, eli bugia, haavoittuvuutta tai huonoa koodisuunnittelua, löytyi.

5.3 Retire.js

Kun SonarQube käsitteli raporttien sisältöä havaintopohjaisesti, Retire.js taas ryhmittelee havainnot tiedostojen mukaan. Sen JSON-raportti luettelee jokaisen tuloksen jokaista läpikäytyä tiedostoa kohti.

Raportti kertoo, mikä riippuvuus on haavoittuvainen, mikä on sen versio ja mitä haavoittuvuuksia siihen liittyy. Siitä löytyy uhkan tunnistenumero ja linkit, jotka kertovat lisätietoja havainnosta. Lisäksi se luokittelee sen vakavuuden asteikolla “low”, “medium” ja “high”. Retire.js on pienempi työkalu, joten se tuottaa huomattavasti pienemmän raportin kuin muut.

5.4 Dependency-Check

Testiraportiltaan Dependency-Check on yksi kookkaimmista. Syynä siihen ei ole havaintojen määrä, vaan todisteiden listaaminen ja viitteet haavoittuvaisiin versioihin. Sen muoto on XML, ja pituus noin 85 tuhatta riviä, mikä on pisin kaikista raporteista.

Raportti alkaa listaamalla käytetyt lähteet, joista löytyy NVD:n haavoittuvuustiedot vuosilta 2002-2019. Sen rakenne pohjautuu riippuvuuksiin. Jokainen löydetty riippuvuus käsitellään yksi kerrallaan, liittyi siihen havaintoa tai ei, mikä on yksi syy raportin pituuteen. Niistä listataan todisteet, eli viitteet riippuvuuden käytöstä, jotka ilmoittavat sen sijainnin, sekä kuinka varmasti kyseessä on pätevä haavoittuvuus. Lisäksi raportista löytyy tieto toisistaan riippuvista riippuvuuksista.

Riippuvuuden haavoittuvuuksista saadaan CVE- ja CWE-tunnistenumero, vakavuusaste, CVSS-arvio ja kuvaus. Myös linkit eri lähteisiin, joista selviää lisätietoja, on sisällytetty haavoittuvuusosioon. Siinä listataan myös kaikki ohjelmat, joita löydetty haavoittuvuus koskee. Luettavuuden kannalta tämä on välillä ongelmallista, ja esimerkiksi osa haavoittuvuuksista löytyy noin neljästä tuhannesta eri Google Chrome -selaimen versiosta, mikä tarkoittaa saman verran lisärivejä raporttiin. Kuvassa 14 on esimerkki. Tällaisia tuhansien rivien rykelmiä saattaa olla useampi peräkkäin.

```
<software>cpe:/a:google:chrome:6.0.418.7</software>
<software>cpe:/a:google:chrome:5.0.332.0</software>
<software>cpe:/a:google:chrome:6.0.418.8</software>
<software>cpe:/a:google:chrome:6.0.418.5</software>
<software>cpe:/a:google:chrome:11.0.684.0</software>
<software>cpe:/a:google:chrome:6.0.418.6</software>
<software>cpe:/a:google:chrome:6.0.418.3</software>
```


Kuva 14. Jokainen rivi kuvaa tiettyä Google Chrome -versiota, johon haavoittuvuus pätee. Tällaisia rivejä on useita tuhansia, eikä niistä ole hyötyä tässä työssä. Todennäköisesti nämä ylimääräiset rivit hidastavat raportin koneellista käsittelyä.

Tässä raportissa vakavuusaste voi olla arvoltaan “low”, “medium”, “high” tai “critical”, mikä heijastuu havainnon CVSS-arvosanasta.

5.5 Zed Attack Proxy

ZAPin rajapinta osaa palauttaa sekä XML- että JSON-muotoisen raportin, ja tässä työssä valittiin jälkimmäinen sen helpon käsiteltävyyden takia. Havainnot on lajiteltu skannatun verkkosivun perusteella, ja ne erotellaan IP-osoitteen ja portin perusteella. Niistä mainitaan perustiedot, kuten osoite, portti sekä se, onko yhteys ollut SSL-salattu.

Havaintotiedoista löytyvät sen nimi, numeraalinen riskiarvo, luotettavuus ja kuvaus. Riski arvioidaan asteikolla 0-3, ja luotettavuus asteikolla 0-4. Numerot kääntyvät arvoiksi “info”, “low”, “medium” ja “high”, sekä “false positive”, “low”, “medium”, “high” ja “user confirmed” [51]. Kuvaus sisältää lyhyen tekstin havainnosta. Raportissa luetellaan myös jokainen sivun instanssi, josta mahdollinen uhka löytyy. Ne erotetaan toisistaan sivun hakemiston, käytetyn HTTP-metodin ja parametrin perusteella. Parametri voi olla tyypiltään HTTP-otsikko, URL:ssa oleva parametri tai sivulta löytyvä riippuvuus, jonka löytyessä liitetään sivun tietoihin todisteet sen käytöstä. Todisteena voi olla esimerkiksi koodirivi, jolla riippuvuus otetaan käyttöön. Mainittakoot, että sen sisältö saattaa rikkoa raportin rakennetta niin, ettei sitä tunnisteta enää missään ohjelmointiympäristössä JSONiksi, kuten kuvassa 15 näkyy. Tätä ongelmaa ei ole, kun raportti haetaan suoraan rajapinnasta kovalevyltä lataamisen sijaan.

```
"evidence":"<script src=\\"//cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js\\"><\\/script>"}],"count":'
```

Kuva 15. Väärin käsitelty lainausmerkki rikkoo loppuosan raportista, eikä sitä voi enää jäsentää JSON-jäsentimellä.

Raportin loppupuolelta löytyy ehdotus havainnon korjaamiseksi ja lisä- sekä tunnistetietoja. Lisätiedot ovat kuvausta pidempi kertomus havaitun parametrin ja havainnon tarkoituksesta, ja ne ovat ympäröity ”<p>”-elementillä, kuten kaikki muukin pidempi tekstiaineisto tässä raportissa. Havaintoon liittyy lisätietojen ohessa linkkejä, ja tunnistetiedoista annetaan WASC- ja CWE-tunnisteet.

5.6 OpenVAS

OpenVAS-manageri kokoaa XML-raportin, jossa skannauksessa löytyneet havainnot luetellaan yhtenä listana. Testiraportista löytyy lisäksi tämän työn kannalta hyödytöntä tietoa monta tuhatta riviä.

Havainnot on oletuksella rajoitettu tuhanteen jokaista raporttia kohti, joten siinä myös kerrotaan, minkä osan kaikista havainnoista raportti kulloinkin sisältää. Toimintaperjaate on siis sama kuin SonarQuben raportin sivutiedoissa. Havainnoista annetaan perustiedot, kuten sen nimi ja kohteen sijainti, ja onnistuneen NVT:n tiedot. Koska OpenVAS skannaa verkon kautta, on sen kohde määritelty verkko-osoitteen ja portin perusteella, ja kohteeseen saattaa kuulua yksi tai useampi tietokone. NVT:n tiedot taas liittyvät haavoittuvuuteen, jota se testaa. Raportti määrittelee sen tyypin, joita ovat esimerkiksi palvelunesto, tiedonsiirtoprotokolla ja verkkopalvelun väärinkäyttö, ja antaa CVSS-arvion sen uhkaavuudesta. Myös tunnistetiedot, kuten CVE- ja Bugtraq-tunnisteet, sekä liittyvät linkit ilmoitetaan.

Yksi luettavuuden ja käsiteltävyyden kannalta ongelmallisimpia kohtia haavoittuvuuksien tiedoissa on elementti “tags”. Siinä on koottu yhdeksi merkkijonoksi seuraavat asiat:

- CVSS-tiedot
- ratkaisuehdotus
- kooste
- ratkaisun tyyppi

- havainnon tarkkuus
- kuinka havainto löydettiin
- tarkempi kooste
- vaikutukset
- kohdekomponentti.

Nämä on erotettu toisistaan pystyviivalla [kuva 16]. CVSS-tiedot ovat lyhenteitä oikeista nimistään ja arvoistaan. Esimerkiksi merkkijono ”AV:N” tarkoittaa hyökkäysvektorin, “AV” eli “Attack Vector”, olevan verkko, “N” eli “Network” [52].

Ratkaisuehdotuksessa kerrotaan suositetut toimenpiteet haavoittuvuuden ratkaisemiseksi, ja sen tyyppi kertoo, onko ehdotettu ratkaisu pelkkä tilapäinen korjaus, ongelman lieventäminen vai ohjelman kehittäjän oma korjaus. Myös kohdekomponentti-kenttä on hyödyllinen. Siitä näkee, jos kyse on esimerkiksi ohjelman vanhentuneesta versiosta. Kooste on pinnallinen kuvaus ongelmasta, eli esimerkiksi siitä, mitä haittaa siitä on ja mistä se johtuu, ja sitä tarkentaa “tarkempi kooste” -kenttä, joka antaa vielä laajemman käsityksen aiheesta. Raportin loppupuolella on erillisessä elementissä vielä kolmas kuvaileva kohta, joka on kaikista lyhyin.


```
<tags>cvss_base_vector=AV:N/AC:H/Au:N/C:P/I:P/A:P|summary=This host is installed with PHP and is prone
to Man-in-the-middle attack vulnerability.|vuldetect=Checks if a vulnerable version is present on the target host.|insight=The following flaws exist:

- The web servers running in a CGI or CGI-like context may assign client request proxy header values to internal
HTTP_PROXY environment variables.

- &apos;HTTP_PROXY&apos; is improperly trusted by some PHP libraries and applications

- An unspecified flaw in  the gdImageCropThreshold
function in &apos;gd_crop.c&apos; in the GD Graphics Library.|impact=Successfully exploiting this issue may allow
remote, unauthenticated to conduct MITM attacks on internal server subrequests
or direct the server to initiate connections to arbitrary hosts or to cause a
denial of service.|affected=PHP versions 5.x through 5.6.23 and 7.0.x through 7.0.8 on Linux|solution=Update to PHP version 5.6.24 or 7.0.19.|solution_type=VendorFix
```


Kuva 16. Tags-elementti on oudosti rakennettu ja sisältää useita eri tietueita tekstimuodossa. Osa sen sisällöstä on lisäksi URL-koodattu.

Raportti listaa havainnon CERT-tunnisteet erikseen, ja niiden perusteella saa lisätietoa siihen liittyvistä ohjelmista, alustoista ja haavoittuvuuksista. OpenVAS määrittää havainnon uhan asteikolla “debug”, “log”, “low”, “medium” ja “high”.

Havaintojen luettelun jälkeen testiraportissa alkaa yksityiskohdista kertominen, ja se jatkuu noin 13 000 riviä. Ne sisältävät tietoa skannausprosessista, kuten mitkä testit ajettiin tuloksetta. Tietoon sisältyy ajetun testin tunniste ja ilmoitus, ettei haavoittuvuutta löytynyt. Kuvassa 17 on yksittäinen yksityiskohta-elementti.

```
<detail>
    <name>EXIT_CODE</name>
    <value>EXIT_NOTVULN</value>
    <source>
        <type>nvt</type>
        <name>1.3.6.1.4.1.25623.1.0.108446</name>
        <description></description>
    </source>
    <extra></extra>
</detail>
```

Kuva 17. Koko skannausprosessin kartoittamiseksi raporttiin on listattu myös ajetut epäonnistuneet testit. Name-elementti kertoo, mikä testi oli kyseessä, ja value-elementti lopputuloksen.

OpenVAS-testiraportti on kankea, ja se tekee asioita liian monimutkaisesti. Sen käsittely koneellisesti ei ole vaikeaa, mutta silti muita raportteja vaivalloisempaa.

5.7 Retina

Retinan XML-raportti on rakenteeltaan yksinkertainen. Siinä ei ole monia kerroksia sisäkkäisiä XML-elementtejä, vaan kaikki on jäsennelty testien perusteella jokaista skannattua IP-osoitetta kohden. Sen avulla kohteesta saadaan tietoon muun muassa sen BIOS- ja DNS-nimi, MAC-osoite ja käyttöjärjestelmä.

Raportointi perustuu testien kautta löytyneisiin haavoittuvuuksiin, aivan kuin OpenVAS-raportissa. Tässä raportissa on kuitenkin vain yksi havaintoa kuvaileva kenttä, joka kertoo siitä muutamalla virkkeellä. Retinan raportti on myös ainoa, joka ottaa kantaa PCI-standardin noudattamiseen. Siitä luetellaan havainnon taso, syy ja onnistuminen. Tämän lisäksi ilmoitetaan sen protokolla, portti, CVSS-arvosana, korjausehdotus ja testin tiedot, jotka tarkentavat, millä testitiedolla saatiin kulloinenkin tulos. Annettuja tunnistetietoja ovat CVE-, CCE- ja CWE-tunnisteet.

6 Rakenne

Raportin uuden rakenteen määrittely oli tehtävä ensimmäisenä, sillä sen tietoja ei voi muuttaa uuteen muotoon ennen sellaisen suunnittelua. Sitä varten täytyi perehtyä eri raporttien sisältöön, jotta näkisi niiden tietojen eroavaisuuksia ja yhteneväisyyksiä. Havaintoja kuvailevat kohdat ovat niiden tärkein osuus, ja kaikissa raporteissa ne voi vielä jakaa sen ominaisuuksia, sijaintia ja vaikutuksia kuvaaviin tietoihin.

Ominaisuuksista kertovat raportin nimi-, kuvaus-, tunniste-, referenssi- ja vakavuuskentät. Useammat kuvauskentät voidaan yhdistää, ja tunnistekentillä tarkoitetaan viitettä mihin tahansa olemassa olevaan listaukseen havainnosta, kuten CVE- ja CWE-luettelot. Referenssit, eli linkit, taas johtavat tunnisteen sivuille, blogikirjoituksiin tai yleisiin keskusteluihin havainnosta. Ne on hyvä sisällyttää tietorakenteeseen niiden avulla saatavien lisätietojan takia. Seuraavassa kuvassa ovat Retire.js-tunnistaman haavoittuvuuden lähdelinkit, joiden avulla voi esimerkiksi arvioida sen todenperäisyyttä. Vakavuuskentät koostuvat useimmiten sanallisesta arviosta, jonka asteikko eroaa raporttien välillä, ja joskus CVSS-arvosanasta.

jquery	←	1.6.3	https://nvd.nist.gov/vuln/detail/CVE-2011-4969 http://research.insecurelabs.org/jquery/test/ https://bugs.jquery.com/ticket/9521

Kuva 18. Retire.js jquery -kirjaston 1.6.3-version linkit näyttävät olevan luotettavista lähteistä. [30]

Havainnon sijainti riippuu sen löytäneen työkalun toiminnasta. Jos se on tyypiltään SAST, voidaan sijainniksi ilmoittaa hakemistopolku skannattuun tiedostoon ja oikea rivi tai tietty riippuvuus. DAST-työkaluissa taas käytetään hakemistopolun sijaan verkko- tai IP-osoitetta sekä porttia. Osa haavoittuvuuksista voidaan rajata tiettyyn HTML- tai URL-parametriin.

Raportteihin liittyy välillä havainnon vaikutuksia esitteleviä kenttiä, jotka kertovat muun muassa, mitä vahinkoa se voi aiheuttaa. Osassa ohjeistetaan suoraan havainnon ratkaisuun. Jos näitä kenttiä ei ole valmiiksi saatavilla, voi tietoturva-asiantuntija täydentää ne

myöhemmin. Rakenteeseen täytyy lisäksi sisällyttää hallinnointiin liittyvää tietoa, jota on esimerkiksi havainnon omistaja, tila, loppuratkaisu, kommentit ja löytäjä.

Rakenteen luokkarakenteen suunnittelun suurin kysymys oli, luodaanko raporteille yhteinen luokka vai jaetaanko se useampaan osaan raporttien tyyppien perusteella. Skannerit tekevät eri asioita, mutta osa niistä muistuttaa toiminnaltaan ja tuloksiltaan toisiaan. Esimerkiksi Retire.js ja Dependency-Check testaavat molemmat riippuvuuksia ja OpenVAS sekä Retina verkkohaavoittuvuuksia. Voisi siis olla hyvä idea jakaa havainnon rakenne tyyppien mukaan niin, että tyyppikohtaiset luokat täydentävät yleistä rakennetta [Liite 1, Kuva 23]**.** Pelkona on, että jos paljon tyyppikohtaista dataa yhdistäisi yhteen luokkaan, siihen syntyisi paljon tarpeettomia kenttiä niiden erityisyyden takia.

Suurin osa tietueista on kuitenkin yhteisiä, ja osan kentistä voisi yhdistää. Esimerkiksi URL- ja Path-kentät tarkoittavat lähes samaa asiaa, ja komponentti sekä versiotiedot voidaan sisällyttää toiseen kenttään, kuten kuvaukseen. Jos kuitenkin halutaan säilyttää mahdollisuus suodattaa ja järjestää havaintoja niiden perusteella, ne pitää jättää erillisiksi kentiksi.

Vaikka skannerit toimivat eri tavoin, ne tuottavat melkein samanlaista tietoa, eikä tietojen jakaminen aliluokkiin selventäisi rakennetta [liite 1, kuva 23]. Yhden luokan käyttäminen on myös linjassa formaattien yhtenäistämisen, tämän työn aiheen, kanssa. Mitä enemmän aliluokkia luodaan, sitä erilaisimmiksi eri tyyppisten työkalujen rakenteet muuttuvat, eikä niiden vertailu olisi enää yhtä suoraviivaista. Lopputuloksena syntyy siis yksi havaintoluokka, jossa yhdistettiin osa tyypitettyjen luokkien kentistä. Samanlaiseen ratkaisuun oli päädytty DefectDojossa. Sen rakenteeseen verrattuna tämä on kuitenkin paljon kompaktimpi. DefectDojolla rakenteeseen kuuluu 60 riviä kenttiä. Tässä rakenteessa on vain 21 [7]. Se johtuu erilaisesta havaintojen hallinnoinnin suunnittelusta ja raporttien eri tietueiden priorisoinnista.

Jokainen havainto sisältää tunnistetietoja mahdollisesta haavoittuvuudesta. Ne vaihtelevat paljon työkalujen välillä, mutta kaikki noudattavat kuitenkin muotoa, jossa ensimmäisten kirjainten avulla voi selvittää tietokannan tai listan, josta tunniste on peräisin. Jos jokaiselle alkuperälle, kuten CWE, CVE, CCE ja WASC, luotaisiin oma kenttä, niitä jäisi usein käyttämättä, sillä työkalut käyttävät eri lähteitä. Tämän välttääksemme luodaan yksi luokka tunnisteille, jossa on ainoastaan viittaus haavoittuvuuteen sekä tunnisteen

nimi [liite 1, kuva 24]. Myös referenssit muodostavat samanlaisen luokan, sillä ORM-toteutus vaatii monesta-moneen-suhteen niiden kohdalla. Linkkien ja tunnisteiden luokat ovat rakenteeltaan täysin samanlaiset, minkä takia jatkokehitysideana olisi yhdistää ne yhdeksi luokaksi [liite 1, kuva 24]. Tämä luokka sisältäisi vielä tyyppikentän, joka kertoisi, onko alkio referenssi vai tunniste.

Rakenteeseen kuuluu myös joukko esimääriteltyjä vakioarvoja, jotka tallennetaan listoihin ja hakurakenteisiin. Ne ovat suurimmaksi osaksi muuttumattomia arvoja, kuten havainnon vakavuus, tila, lopputulos, ohjelmointikieli sekä käytetty skanneri. Määrittely tehdään, jotta tallennettava tieto pysyisi yhdenmukaisena, eikä esimerkiksi arvojen kirjoitusasussa olisi ristiriitoja. Koska arvot eivät muutu tai päivity, ne kannattaa sisällyttää osaksi koodia, koska se nopeuttaa niiden käyttämistä verrattuna esimerkiksi tietokantataulun käyttämiseen. Lisäksi tallennetaan kartoitus raporttien omista vakavuusasteikoista esimääriteltyyn vakavuusasteikkoon, joka on suunniteltu Accenturen käyttötarkoitukseen sopivaksi arvoilla “info”, “low”, “medium” ja “high”. Lista- ja hakurakenteiden sijaan kokeiltiin myös selvempää “Enumeraatio”-rakennetta, mutta se vaatii aina hakurakeenteen tapaan avaimen ja siihen liittyvän arvon, mikä ei olisi järkevää listarakennetta korvatessa. Seuraavana on esimerkki koodiin tallennetuista vakioarvoista.

```
2   LANGUAGE = ["java","javascript"]
3   STATUS = ["open","closed","waiting"]
4   RESOLUTION = ["fixed","wontfix","false positive","removed"]
5   TOOL = ["sonarqube","dependencycheck","retirejs","retina","openvas","zap"]
6
7   #Severity related mapping
8   SEVERITY = ["info","low","medium","high"]
9   low_severity = SEVERITY[SEVERITY.index("low")]
10  high_severity = SEVERITY[SEVERITY.index("high")]
11  medium_severity = SEVERITY[SEVERITY.index("medium")]
12  info_severity = SEVERITY[SEVERITY.index("info")]
13  REPORT_SEVERITY = {
14      "blocker" : high_severity,
15      "critical" : high_severity,
16      "major" : medium_severity,
17      "minor" : low_severity,
18      "1" : low_severity,
19      "2" :  medium_severity,
20      "information" : info_severity,
21      "log" : info_severity,
22      "debug" : info_severity
23      }
```

Esimerkkikoodi 3. Riveillä 2-8 määritellään vakioarvoja listoihin, jonka jälkeen kartoitetaan raporttien vakavuuksien määrittäminen vakiovakavuuksien mukaisesti. Esimerkiksi raporteissa olevat “blocker”- ja “critical”-vakavuudet muutetaan määritelmän mukaiseen “high”-vakavuuteen aina havaintoa luodessa.

Vakioarvojen käyttö tapahtui aluksi manuaalisesti. Joka kerta, kun havainnolle annettiin esimääritetty arvo, täytyi oikeasta listasta hakea oikea arvo. Tämä kuitenkin altistaa huolimattomuusvirheille, sillä esimerkiksi kehittäjä saattaa unohtaa hakea kielen vakioarvon "java" ja määrittääkin havainnon kielen itse "JAVA":ksi . Jotta tätä ei tapahtuisi, hyödynnettiin Djangon "pre_save"-signaalia [12, s. 1136]. Se lähetetään ennen tallennusta, ja sille voi määrätä signaaliin reagoivan käsittelijän, joka tässä tapauksessa automaattisesti vaihtaa tallennettavan havainnon kentät vakioarvoja vastaavaan muotoon, tai vähintään ilmoittaa virheellisestä arvosta ja estää sen tallentamisen. Virhetilanteen voi yleensä välttää päivittämällä vakioarvoja aina, kun uuden työkalun lisää projektiin.

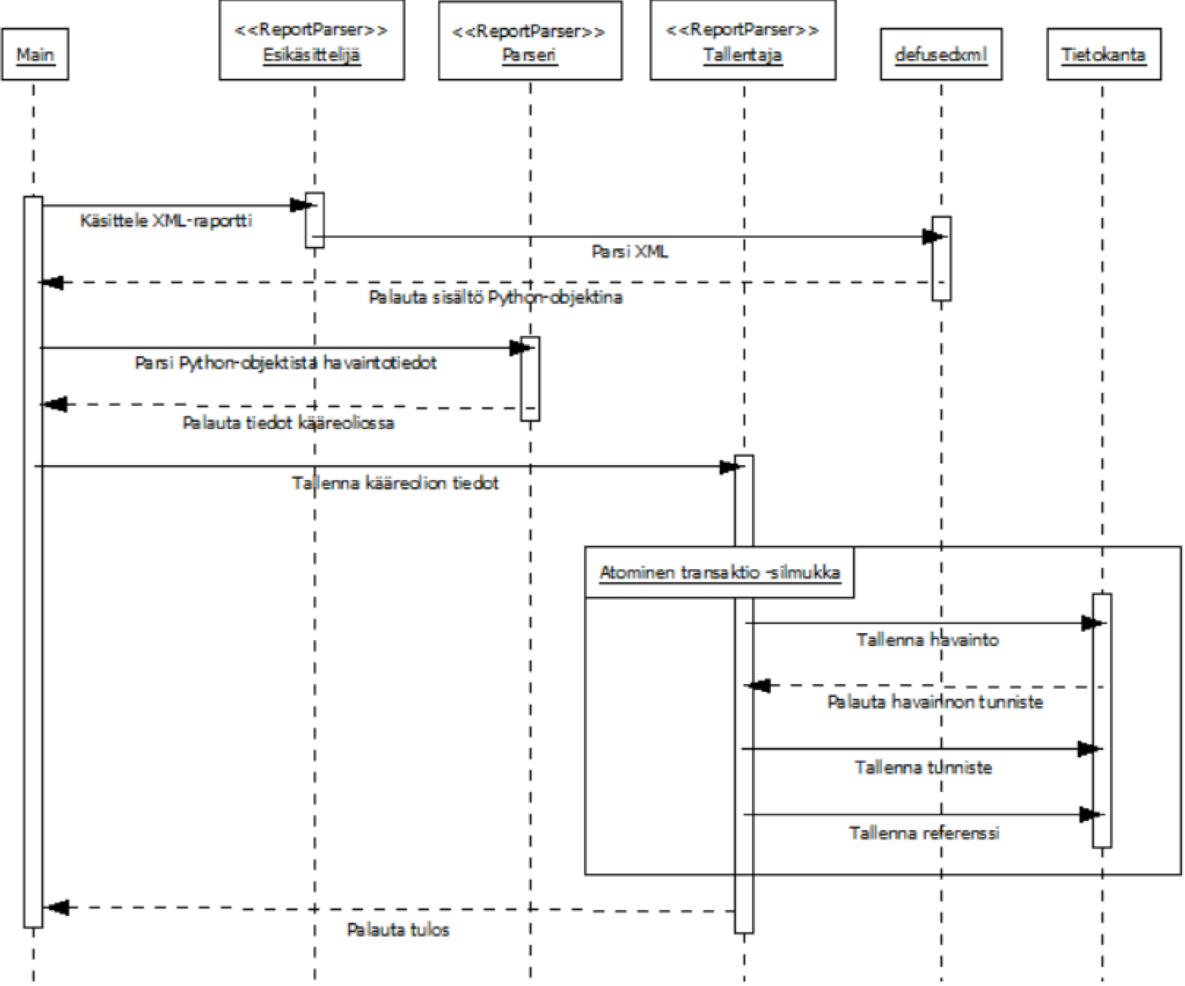
Kun tässä työssä vakioarvot on sijoitettu erilliseen tiedostoon, DefectDojossa taas vakioarvot ja niiden käyttö on ainakin osittain sekaisin muun lähdekoodin kanssa. Esimerkiksi jokaisen raportin vakavuuden kartoitus on ohjelmoitu osaksi sen omaa jäsennintä [6], mikä vaikuttaa selkeämmältä ratkaisulta ja on ehkä järkevää toteuttaa tulevaisuudessa tässäkin projektissa.

7 Jäsentimet

Raporttien tulosten uudelleenmuotoilua kutsutaan tässä työssä jäsentämiseksi, mikä vastaa englannin kielen termiä “parsing”. Ohjelmoinnissa se tarkoittaa tekstimuotoisen rakenteen koneellista ymmärtämistä ja jakamista merkityksellisiin osiin, ja tässä projektissa se saavutetaan käyttämällä valmiita kirjastoja sekä kehittämällä oma jäsennin jokaiselle raportille [53; 54]. Jäsentimellä tarkoitetaan komponenttia, joka suorittaa jäsentämisen.

Pythonin json-kirjasto osaa jäsentää JSON-raportin ja muodostaa siitä Python-objektin [55], ja XML-raportteihin taas käytetään defusedxml-kirjastoa, joka jäsentää niistä helposti käytettävän puurakenteen [56]. Koska XML on ollut pitkään haavoittuvainen ohjelmointikieli, defusedxml pyrkii erityisesti suojaamaan mahdollisilta hyökkäyksiltä esimerkiksi käyttämällä oletuksena tietoturvallisia asetuksia [56]. JSON- ja XML-jäsentimet eivät tietenkään osaa tunnistaa raporteista aiemmin määriteltyä muotoa, joten tarvitaan komponentti, joka jatkojalostaa kirjastojen jäsentämät tulokset vastaamaan sitä.

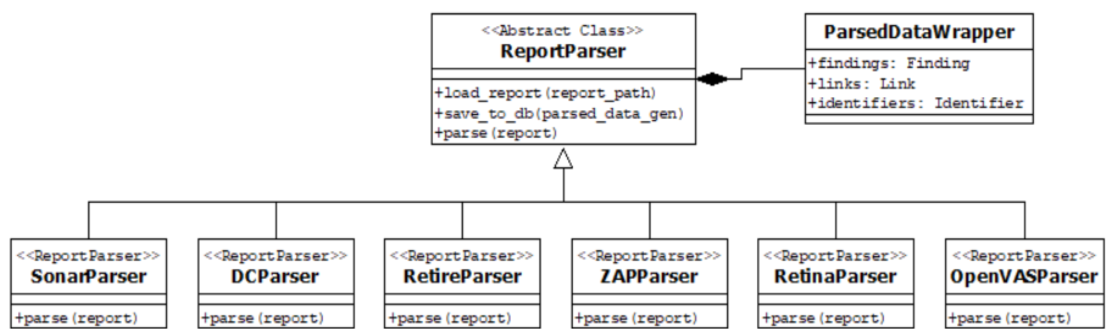
Tämä komponentti koostuu raportin esikäsittelijästä, jäsentimestä ja tallennusoperaatiosta ja on nimeltään ReportParser. Se on abstrakti luokka, joka sisältää yleisen toteutuksen havaintojen esikäsittelylle ja tallentamiselle sekä abstraktin funktion raportin jäsentämiselle, joka on toteutettu siitä periytyvissä jäsenninluokissa. Esikäsittelijän tehtävänä on tunnistaa raportin tiedostomuoto, joka luetaan tiedoston päätteestä, ja valita sen perusteella joko json tai defusedxml sen lataamiseksi ja Python-objektiksi muuttamiseksi. Objekti annetaan ReportParserista periytyvän luokan jäsennettäväksi, joka lukee siitä halutut arvot ja luo siitä aikaisemmassa luvussa määrittelemämme havainto-, tunniste- sekä linkki-objektin. Tallennusoperaatio tallentaa lopuksi kaiken tietokantaan. Prosessissa on tarkoituksella eristetty raportin lataaminen, jäsentäminen ja tallentaminen toisistaan eri funktioihin. Kuvassa 19 on vielä sekvenssikaavio prosessista, jotta siitä saisi paremman mielikuvan.



Kuva 19. Prosessi raportin esikäsittelystä sen tallentamiseen. Main on komponentti joka pitää hallussaan raporttia ja kutsuu ReportParseria toteuttavaa luokkaa sen käsittelemiseksi. Oikeassa alakulmassa oleva silmukka selitetään myöhemmässä luvussa, kun tallennusoperaatioon perehdytään tarkemmin.

ReportParserista periytyy jokaiselle raportille oma luokka, joka toteuttaa siihen sopivan jäsentimen [kuva 20]. Yleinen idea on, että se tunnistaa esikäsittelyssä syntyneestä objektista tarvittavat tietueet ja palauttaa kääreolion, johon tiedot havainnosta, tunnisteista ja linkeistä on sijoitettu väliaikaisesti. Jäsentimen aikana objektia käsitellään silmukoissa, joissa sen rakenteesta haetaan tietoja. Silmukoiden määrä ja järjestys heijastuvat suoraan raportin omasta rakenteesta. Esimerkiksi Retire.js:n havaintotietoihin käsiksi pääseminen vaatii ensin silmukan jokaiselle skannatulle tiedostolle, ja sen jälkeen silmukan tiedoston havaintoja varten. Havaintoihin voi liittyä monta tunnistetta ja linkkiä, joten tarvitaan vielä kolmas silmukka niitä varten. Lopputuloksena käydään läpi tunnisteet ja linkit jokaista havaintoa kohden, joita käsitellään per tiedosto. SonarQuben jäsennin taas käy

kaiken läpi yhdessä silmukassa, koska tämä on myös sen raportin rakenne. Suorituskyvyltään jäsentimet ovat siis täysin riippuvaisia raportin muodosta.



Kuva 20. Tässä ovat kaikki ReportParserista periytyvät jäsentimet, sekä sen sisäluokka, ParsedDataWrapper, jota käytetään jäsennettyjen tietojen käärimiseen.

Silmukoiden lisäksi ainoat erot raporttien jäsentämisessä ovat jäsentimien apufunktiot. Niitä ei näy jäsenninfunktion ulkopuolelle, kuten edellisestä kuvasta voi huomata, eli ne on kapseloitu sen sisälle, ja ne auttavat tietyn raportin käsittelyä yleisellä tasolla. Esimerkiksi Dependency-Check tarvitsee hakujen yhteydessä nimiavaruuden jokaisen XML-elementin etuliitteeksi, jonka takia sen ja elementin yhdistämiselle on tehty apufunktio. SonarQube-raportin jäsentämiseen taas liittyy sääntökentän hakeminen avaimen perusteella, mikä on myös eristetty omaksi sisäfunktiokseen. Tällaiset toiminnallisuudet eivät hyödytä muita jäsentimiä, joten niiden näkyvyys rajoitetaan niin vähäiseksi kuin mahdollista. Seuraavaksi kerrotaan toteutetuista jäsentimistä ja tallennusoperaatiosta tarkemmin, ja luvun lopussa verrataan toteutuksien suorituskykyjä.

7.1 Listatoteutus ja tallentaminen

Ensimmäisenä syntynyt ratkaisu raporttien jäsentämiseen oli toteutettu hyväksikäyttämällä väliaikaisia listoja. Koska koko jäsentämisen tulokset palautetaan yhdessä jäsentämisprosessin lopussa, täytyy kaikki havainnot, tunnisteet ja linkit ottaa väliaikaisesti talteen ennen sitä. Jokaisen havaintoja käsittelevän silmukan jälkeen jäsennetyt tiedot säilötään listoihin niin, että listan indeksi vastaa aina yhtä havaintoa. Toisin sanoen havaintolistan ensimmäisessä indeksissä on ensimmäinen havainto, tunnistelistan ensim-

mäisessä indeksissä siihen liittyvät tunnisteet ja referenssilistan ensimmäisessä indeksissä siihen liittyvät linkit. Kun kaikki tulokset on jäsennetty, listat talletetaan kääreluokan edustajaan, jokainen omaan kenttäänsä. Kääreolio palautetaan ja se voidaan tallentaa tietokantaan.

Seuraavana oleva esimerkkikoodi on esimerkki listatoteutuksen jäsentämisprosessista. Esimerkkikoodin riveillä 25-26 kaikki yhteen havaintoon liittyvät referenssit kerätään links-nimiseen listaan, joka tallennetaan linksList-listan jatkoksi. Rivillä 28 kaikki havainnot taas lisätään findings-listaan samaan indeksiin, mihin sen linkit menivät linksList-listassa. Yhteisen indeksin takaa listojen käsittely samassa silmukassa. Lopussa listat kääritään palautettavaan ParsedDataWrapper-olioon.

```
17  class DCParser(ReportParser):
18      def parse(self,report):
19          ...
20          for d in dependencies:
21              ...
22              for i in issues:
23                  ...
24                  references = find_xml_element(i,"reference",True)
25                  for ref in references:
26                      links.append(find_xml_element(ref,"url").text)
27                  linksList.append(links)
28                  findings.append(finding)
29          return ReportParser.ParsedDataWrapper(identifiers,linksList,findings)
```

Esimerkkikoodi 4. DCParserin rakenne koostuu jäsentämisfunktiosta, joka käy läpi Dependency-Checkin raportin.

Tallennusfunktio ottaa vastaan käärityt arvot ja purkaa ne taas erillisiksi listoiksi. Koska havaintoja saattaa olla satoja ja tietokantakutsut ovat raskaita, funktio täytyy toteuttaa tehokkaasti. Esimerkiksi tietokantaoperaatioita ei tulisi käyttää silmukoissa sen skaalautumattomuuden vuoksi, lukuisien tallennusoperaatioiden tekeminen silmukassa voisi pysäyttää koko ohjelman toiminnan useammaksi sekunniksi. Django tarjoaa onneksi mahdollisuuden tallentaa useamman kohteen kerralla listassa [12, s. 1193], joten yksi vaihtoehto olisi tallentaa ensin kaikki havainnot kerralla, ja sen jälkeen niihin liittyvät tunnisteet ja linkit, joita kutsutaan tässä työssä havainnon liitteiksi. Tämä veisi kolme tietokantakutsua. Mutta linkitetäänkö liitteet havaintoon ennen vai jälkeen sen tallentamisen?

Ensimmäisenä tulisi mieleen asettaa liitteet etukäteen, koska jotkut ORM-järjestelmät tallentaisivat ne havainnon kanssa automaattisesti. Django ei kuitenkaan tue tätä. Se ei itse asiassa salli havainnon tallentamista ollenkaan, jos siihen liittyy tallentamattomia liitteitä, joita taas ei voi tallentaa, jos ne viittaavat tallentamattomaan havaintoon. Tämä johtuu siitä, ettei havainto- ja liiteoliolla ole tunnistetta ennen tallentamista, koska se saadaan tietokannalta [57]. Tunniste taas vaaditaan tallentamisvaiheessa kaikilta viitattavilta olioilta, sillä se vastaa tietokannan viiteavainta, jolla luodaan yhteys tietokantaobjektien välille. Ainoa toimiva vaihtoehto on siis tallentaa havainnot ilman liitteitä, ja liittää ne jälkikäteen.

Havaintojen tallentamisen jälkeen niillä on tunniste, johon sen liitteet voisivat viitata. Django ei kuitenkaan päivitä havainto-olioilleen tietokannan luomaa tunnistetta, jos niitä tallentaa monta kerrallaan listassa, ellei tietokanta ole PostgreSQL [12, s.1193]. Se vaikeuttaa koko tallennusprosessia huomattavasti, sillä olioiden tiedot täytyisi hakea uudestaan tietokannasta, jotta niiden tunniste päivittyisi ja liitteet saataisiin linkitettyä. Tämä eroaa yksittäisen olion tallentamisesta, jossa tunniste päivittyy heti, kun toimenpide onnistuu. Onneksi on vielä yksi vaihtoehto.

Django tukee nimittäin atomisia transaktioita. Transaktio tarkoittaa yhtenä toimenpiteenä suoritettavaa tietokantaoperaatioiden sarjaa, ja atomisuus merkitsee transaktion suoriutuvan ainoastaan, jos koko sarja operaatioita onnistuu [58]. Muulloin se peruutetaan. Djangon kohdalla atomisuus tarkoittaa myös, että jos transaktion operaatiot suoritetaan virheettömästi, Django tekee kaikki operaatiot yhdellä tietokantakutsulla sen lopussa [12 s. 147]. Tämän tiedon valossa silmukassa tallentaminen onkin kannattavaa, toisin kuin alussa luultiin. Riittää, että se tehdään atomisen transaktion sisällä. Sen ansiosta tietokantaa ei ylikuormiteta kutsuilla.

Myös DefectDojon jäsennintoteutukset tallentavat havainnot väliaikaisesti muistiin. Listan sijasta käytössä on kuitenkin hakurakenne, jonka arvoina ovat havainnot, ja avaimina niiden tiivisteet. Tiivisteitä se käyttää kaksoiskappaleiden poistamiseen. Lisäksi DefectDojo käsittelee osaa raporteista eri muodossa, kuten HTML ja CSV. [6.]

Päästään siis toimivaan tilanteeseen, jossa havainnot tallennetaan yksi kerrallaan, ja jokaisen jälkeen tallennetaan siihen liittyvät liitteet. Kaikki tiedot löytyvät tietokannasta juuri niin kuin pitää, joten tätä voisi luonnehtia työn vähimmäistoteutukseksi.

7.2 Generaattoritoteutus

Listatoteutuksen jäsentämisprosessin tehokkuutta ja nopeutta voisi parantaa. Sisällön tallentaminen listoihin, ja niiden kuljetus funktiosta toiseen vie muistia ja kestää kauan. Kaiken lisäksi sisäkkäiset for-silmukat skaalautuvat huonosti, mihin löytyy muutama ratkaisu: NumPy -taulut, “Map”, “List comprehension” sekä generaattorit. Ensimmäinen moduuli kuuluu osaksi Numerical Python -kirjastoa, ja viimeiset kolme on sisäänrakennettu Pythoniin.

Tiedon hakeminen NumPy-tauluista on huomattavasti nopeampaa verrattuna tavalliseen for-silmukkaan, joissakin tapauksissa ero saattaa olla satakertainen [3]. Sen käyttö sallii sen omien hakufunktioiden ja -rakenteiden hyödyntämisen. Siitä ei ole valitettavasti hyötyä tässä työssä, sillä jäsentimien on pakko iteroida jokainen havainto yksi kerrallaan ja melkein kaikki sen kentistä, mihin NumPy ei tarjoa oikoteitä.

Map-funktion tarkoitus on suorittaa funktio jokaiselle listan alkiolle, ja palauttaa ne uuteen listaan muokattuna. Se on kuitenkin hitaampi kuin List comprehension -rakenne, joka tekee saman asian [3; 59]. List comprehension on syntaksiltaan lyhyt silmukka, joka toimii tehokkaammin ja on paljon tiiviimpi syntaksiltaan kuin for-silmukka [3; 60]. Sillä voi luoda helposti uusia listoja tai muokata nykyistä, ja osa pienistä tunnisteiden ja linkkien käsittelyyn liittyvistä for-silmukoista korvattiin sillä. Kookkaampien listojen korvaamisen kanssa ongelmaksi koituivat iteraattorit, jotka palauttavat ainoastaan kerran kulloinkin iteroitavan olion. Sen takia koko jäsentämisosuus pitäisi toteuttaa yhdessä List comprehension -rakenteessa, mikä on hankalaa sen syntaksin takia. Viidennen esimerkkikoodin rivillä 27 on esimerkki tästä syntaksista.

Generaattorit voisivat tehdä aiemmasta toteutuksesta tehokkaamman [60; 61]. Ne ovat funktioita, jotka muistavat tilansa, ja jatkavat aina suoritusta jäämästään kohdasta. Ne ovat myös iteraattoreita, jonka jokainen kutsu palauttaa generaattorin seuraavan alkion. Tästä on suuri hyöty; sen sijaan, että talletetaan havaintoja ja sen liitteitä erillisiin listoihin etukäteen, ne voidaan generoida tallennuksen yhteydessä. Generaattorilauseke on muistinkäytöltään List comprehensiota tehokkaampi, sillä se ei säilytä jatkuvasti koko listaa muistissa, vaan ainoastaan yhden osan kerrallaan [61]. Generaattorien avulla haa-

voittuvuuksia ei tarvitsisi tallettaa listoina muistiin, vaan ne voidaan luoda tarpeen mukaan, minkä takia jäsentimien ja tietokantafunktion toiminta kuuluisi nopeutua merkittävästi. Generaattoreiden kanssa täytyy olla hyvin tarkka niiden kontekstista. Jos luodaan generaattori sisennettyyn silmukkaan, sen ulkopuolelle, kuten ylemmälle silmukalle, jäävät arvot eivät päivity sen eri suorituskerroilla. Siksi jokainen silmukka tulee sisällyttää generaattorin kontekstiin [esimerkkikoodi 5]. Niihin liittyy myös generaattorilauseke, joka on syntaksiltaan kuin List comprehension, mutta hakasulut korvataan tavallisilla suluilla. Se on yksinkertaistettu silmukkageneraattori, eli sekvenssi, josta generoidaan pyynnöstä sen seuraava osa [62]. Esimerkkikoodissa kuusi on vielä esimerkki generaattorin käyttämisestä tallennuksen yhteydessä.

```
17  class DCParser(ReportParser):
18      def parse(self,report):
19          ...
20          def parser_data_generator(root):
21              ...
22              for d in dependencies:
23                  ...
24                  for i in issues:
25                      ...
26                      references = find_xml_element(i,"reference",True)
27                      [links.append(find_xml_element(ref,"url").text) for ref in references]
28                      yield ReportParser.ParsedDataWrapper(finding,identifiers,links)
29          return parser_data_generator(root)
```

Esimerkkikoodi 5. DCParserin jäsentämisfunktio muuttui rakenteeltaan täysin. Sen aiempi toiminnallisuus siirtyi parser_data_generator-generaattorin sisälle, jonka se nyt palauttaa. Generaattori luovuttaa silmukassa yhden kääreolion jokaisella kutsukerralla, minkä huomaa yield-termistä rivillä 28.

Generaattorien kanssa uuden tietoturvaskannerin tuen lisääminen ei vaadi paljon vaivaa. Täytyy ainoastaan selvittää havainnon tietojen hakeminen sen raportin rakenteesta, ja toteuttaa se uuden jäsentimen generaattorissa. ORM-toteutukseen ei tarvitse koskea, mutta staattisia vakioarvoja, kuten haavoittuvuuden vakavuusasteen hakurakenteita pitää tarvittaessa päivittää.

Raporttien jäsentimet muuttuivat siis rakenteellisesti. Sen sijaan, että sellainen palauttaisi suuren kääreolion kaikkine listoineen, palautetaankin yksi generaattori, eli sekvenssi jokaisesta tarvittavasta havainnosta ja liitteestä. Ne palautetaan vieläkin hyödyntämällä kääreluokkaa, mutta listojen sijaan jokaiseen havaintoon liittyvät tiedot kääritään yksittäin erikseen. Lisäksi tietokantafunktio muuttuu vastaamaan uutta toimintamallia,

seuraavan esimerkkikoodin mukaisesti. Nyt se saa parametrinaan yhden generaattorin, jota iteroimalla hankitaan tallennettavat tiedot.

```
47  @transaction.atomic
48  def save_to_db(self,parsed_data_gen):
49      for parsed_data in parsed_data_gen:
50          vuln = parsed_data.findings
51          identifiers = parsed_data.identifiers
52          links = parsed_data.links
53          #Save finding
54          ...
```

Esimerkkikoodi 6. Tallennusfunktio generaattorilla toteutettuna saa pelkän generaattorin parametrina. Ensimmäisellä rivillä on Djangon tapa merkitä atominen transaktio.

Seuraavaksi verrataan, kuinka paljon nämä muutokset paransivat suorituskykyä kolmella eri jäsentimellä. Lähtöoletuksena generaattorien kuuluisi parantaa suorituskykyä merkittävästi. Ainoastaan kolmea jäsennintä mitattiin eri versioiden ylläpidon hankaluudesta johtuen.

7.3 Suorituskykyvertailu

Suorituskykyä mitattiin suoritusajan ja muistinkäytön perusteella käyttäen line_profiler- ja memory-profiler-nimisiä Python-kirjastoja [63; 64]. Ensimmäinen näyttää kuluneen ajan, ja jälkimmäinen käytetyn muistin jokaiselle suoritetulle riville. Lisäksi mittaus rajattiin ainoastaan jäsentämiseen ja tallentamiseen liittyviin funktioihin. Mittaus suoritettiin ensin vanhalle listatoteutukselle ja sen jälkeen uudelle generaattoritoteutukselle.

Memory-profileria täytyy käyttää manuaalisesti, kun mitataan generaattoria [65]. Sen mittauskohdiksi valittiin jäsentämisen ja tietokantaan tallentamisen jälkeiset tilanteet. Yksi mielenkiintoinen huomio on myös, etteivät kirjastot näytä generaattorifunktion riveille tuloksia jäsentämisfunktion toimintaa mitatessa, sillä se suoritetaan vasta jälkeenpäin tallennusfunktiossa. Tämän takia täytyy muistaa merkitä myös itse generaattorifunktio mitattavaksi, tai voi virheellisesti luulla nopeuttaneensa ohjelman toimintaa tuhatkertaisesti. Seuraavana on mittaustulokset listaavat taulukot.

Taulukko 2. Sarakkeet ilmoittavat työkalun sekä onko mittauksen kohde jäsentäminen vai tallentaminen. Rivit ilmaisevat suoritusajat sekunteina ja muistinkäytön megatavuina

sekä listatoteutukselle (LT) että generaattoritoteutukselle (GT) ja paljonko niiden prosentuaalinen ero on neljän desimaalin tarkkuudella.

Retire.js	**Jäsentäminen**	**Tallentaminen**
Suoritusaika, LT	0,0012 s	0,1229 s
Suoritusaika, GT	0,0024 s	0,1039 s
Ajan prosentuaalinen muutos	**98,3471**	**-15,4597**
Muistinkäyttö, LT	55,5585 MB	56,9023 MB
Muistinkäyttö, GT	54,3359 MB	55,6757 MB
Muistin prosentuaalinen muutos	**-2,2005**	**-2,1556**

SonarQube	**Jäsentäminen**	**Tallentaminen**
Suoritusaika, LT	0,0246 s	0,1143 s
Suoritusaika, GT	0,0217 s	0,1135 s
Ajan prosentuaalinen muutos	**-11,7886**	**-0,6999**
Muistinkäyttö, LT	56,4648 MB	57,4179 MB
Muistinkäyttö, GT	56,3828 MB	57,2812 MB
Ajan prosentuaalinen muutos	**-0,1452**	**-0,2380**

Dependency-Check	**Jäsentämien**	**Tallentaminen**
Suoritusaika, LT	0,0691 s	16,8520 s
Suoritusaika, GT	0,0665 s	6,7287 s
Ajan prosentuaalinen muutos	**-3,7626**	**-60,0718**
Muistinkäyttö, LT	76,8710 MB	85,5460 MB
Muistinkäyttö, GT	75,4430 MB	83,7260 MB
Muistin prosentuaalinen muutos	**-1,8576**	**-2,1275**

Odotusten vastaisesti ainoa generaattoreilla saatu parannus oli Dependency-Checkin tallennusfunktiossa. Se on huomattavat 10 sekuntia, joka voisi syntyä listojen ja generaattorin iteroinnin vaativuuserosta. Muistinkäytön erot taas ovat lähellä yhtä megatavua kaikissa mittauskohdissa, mikä ei myöskään vastaa odotuksia. Lisäksi muistinkäyttöä tarkastelemalla huomataan, että generaattorioliolle varataan tilaa 32 tavua enemmän

kuin kääreoliolle, joka sisältää kaiken tallennettavan tiedon. Tämä ei mukaile oletusta generaattorien tehokkuudesta ollenkaan.

Näissä testeissä oli käytössä oikea skannaustyökalun tuottama raportti, joista osa oli pienikokoisia. Jotta saataisiin lisätietoa toteutuksien välisistä eroista, kuten kuinka hyvin ne kestävät kuormitusta, ajetaan samat testit vielä kertaalleen, mutta tällä kertaa manuaalisesti muokatuilla epärealistisen suurilla raporteilla. Alkuperäisessä Retire.js-raportissa oli 7 havaintoa, SonarQuben raportissa 248 ja Dependency-Check -raportissa 176. Uusissa raskaammissa raporteissa vastaavat luvut ovat 1507, 3236 ja 2038. Raporttien muokkaus koostui pelkästään olemassa olevien havaintojen monistamisesta. Seuraavana on suorituskykyvertailu paisutetuilla raporteilla.

Taulukko 3. Sama rakenne kuin edellisessä taulussa, mutta tällä kertaa luvut ja toteutuksien erot ovat suurempia raskaista raporteista johtuen.

Retire.js	**Jäsentämien**	**Tallentaminen**
Suoritusaika, LT	0,2623 s	31,9626 s
Suoritusaika, GT	0,4015 s	26,7819 s
Ajan prosentuaalinen muutos	**53,0690**	**-16,2086**
Muistinkäyttö, LT	63,4648 MB	72,6367 MB
Muistinkäyttö, GT	60,8203 MB	70,9179 MB
Muistin prosentuaalinen muutos	**-4,1668**	**-2,3662**

SonarQube	**Jäsentämien**	**Tallentaminen**
Suoritusaika, LT	0,5994 s	2,8435 s
Suoritusaika, GT	0,6710 s	3,5234 s
Ajan prosentuaalinen muutos	**11,9452**	**23,9106**
Muistinkäyttö, LT	71,2265 MB	77,0507 MB
Muistinkäyttö, GT	69,0703 MB	74,5000 MB
Muistin prosentuaalinen muutos	**-3,0272**	**-3,3104**

Dependency-Check	**Jäsentämien**	**Tallentaminen**

Suoritusaika, LT	0,7152 s	137,8950 s
Suoritusaika, GT	0,8453 s	90,9968 s
Ajan prosentuaalinen muutos	**18,1907**	**-34,0100**
Muistinkäyttö, LT	334,8515 MB	344,1484 MB
Muistinkäyttö, GT	332,6367 MB	341,9882 MB
Muistin prosentuaalinen muutos	**-0,6614**	**-0,6276**

Muistinkäytöltään toteutukset ovat melkolailla yhdenvertaisia: generaattoreilla on aina pieni etu, mutta parhaassa tapauksessa se juuri ja juuri ylittää neljä prosenttia. Jokainen jäsentämisfunktio toimii nyt huonommin generaattoritoteutuksessa, vaikka aiemmassa testissä molemmat toteutukset olivat ajallisesti melkein tasavertaisia. Erot ovat kuitenkin sekunnin kymmenesosia, joten tämä tuskin osoittautuu “pullonkaulaksi” tulevaisuudessa. Sen sijaan tallentaminen parantuu Retire.jsn ja Dependency-Checkin kohdalla merkittävästi generaattorien ansiosta, varsinkin jälkimmäinen valmistuu kymmeniä sekunteja nopeammin, mikä on suuri etu - jopa niin suuri, että jo sen takia generaattoritoteutusta voisi pitää listatoteutusta ylivertaisempana.

Jäsentimissä generaattorilla meni hieman pidempään havaintojen luomisessa, mutta pienempien silmukoiden korvaaminen List comprehensioneilla paransi niiden nopeutta noin kolmanneksella. Ohessa on kuva mittaustuloksista.

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    60      2308      22131.0      9.6      0.9      references = find_xml_element(i,"reference",True)
    61     21787     147259.0      6.8      6.0      for ref in references:
    62     19479     163782.0      8.4      6.7          links.append(find_xml_element(ref,"url").text)
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    69      2308      32193.0     13.9      1.3      references = find_xml_element(i,"reference",True)
    70      2308     186478.0     80.8      7.5      [links.append(find_xml_element(ref,"url").text) for ref in references]
```

Kuva 21. Ylemmässä osassa käytetään for-silmukkaa referenssien lisäämiseksi links-listaan DCParserissa, alemmassa sama tehdään List comprehension avulla. Time-sarakkeesta huomaa jälkimmäisen tavan olevan nopeamp, ja ajan yksikkö on 3.7758e-07 sekuntia.

Tietokantaoperaatioita hidastavat kohdat liittyvät liitteiden ja havainnon välisen suhteen luomiseen. Linkkien ja tunnisteiden liittämisessä menee kaksinkertainen aika listatoteutuksessa [kuva 22]. Tämä selittää myös, miksi SonarQuben raportin tietokantafunktio on ainoa, jossa generaattoritoteutus hävisi listoille: sillä ei ole raportissaan yhtäkään liitettä.

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    80      2308       6478.0      2.8      0.0    if linksList:
    81      2308       6698.0      2.9      0.0        links = linksList[i]
    82     39791     306416.0      7.7      0.1        for link in links:
    83     37483     105416.0      2.8      0.0            if link:
    84     37483     162777.0      4.3      0.0                link_obj = None
    85     37483      84049.0      2.2      0.0                try:
    86     37483  109701118.0   2926.7     26.6                    link_obj = Link.objects.get(link=link)
    87                                                      except Link.DoesNotExist:
    88                                                          link_obj = Link(link = link)
    89                                                          link_obj.save()
    90     37483  294400686.0   7854.2     71.5                vuln.links.add(link_obj)
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    69      2308       6277.0      2.7      0.0    if links:
    70     21787     152551.0      7.0      0.1        for link in links:
    71     19479      50479.0      2.6      0.0            if link:
    72     19479      82570.0      4.2      0.0                link_obj = None
    73                                                      #See if link already exists
    74     19479      40335.0      2.1      0.0                try:
    75     19479   57162131.0   2934.6     20.8                    link_obj = Link.objects.get(link=link)
    76                                                      except Link.DoesNotExist:
    77                                                          link_obj = Link(link = link)
    78                                                          link_obj.save()
    79                                                      #Link link to finding
    80     19479  157377294.0   8079.3     57.3                vuln.links.add(link_obj)
```

Kuva 22. Ylempi puoli on DCParserin linkkientallennusprosessi listatoteutuksena, alempi taas generaattoritoteutuksena. Ylivoimaisesti raskain rivi on molemmissa viimeinen, jossa havainnon ja linkin välille luodaan viittaus, ja lähes jokaisella rivillä listatoteutus on puolet hitaampi kuin generaattoritoteutus. Linkit olivat jo valmiiksi tietokannassa, minkä takia niitä ei tarvinnut tallentaa kummassakaan tapauksessa.

On vaikea sanoa, miksi juuri tietty tietokantaoperaatio hidastaa ohjelmaa, varsinkin kun välissä on Djangon oma ORM-rajapinta. Sen optimoiminen itse olisi työlästä ja vaikeaa, mutta on hyvä tietää Djangon tukevan myös pelkän SQL-koodin kirjoittamista sen muun toiminnallisuuden yhteyteen [66]. Helpoin vaihtoehto on keskittyä oman tietokantarakenteen suunnitteluun ja paranteluun sekä operaatioiden käyttämiseen oikein. Esimerkiksi alun perin jokainen linkki ja tunniste luotiin aina uudestaan välittämättä siitä, löytyykö vastaavaa jo tietokannasta. Myöhemmin koodista löytyi virhe, jonka takia jokainen linkki ja tunniste tallennettiin ylimääräisen kerran. Kun näistä päästiin eroon, osassa työkaluja suoritusaika parantui kymmenillä sekunneilla.

8 Yhteenveto

Tavoitteena oli luoda yhteinen rakenne ja jäsentimet valittujen tietoturvaskannerien raporteille, jotta niiden tietoja voisi tulevaisuudessa käsitellä samoilla säännöillä samassa järjestelmässä. Tuotetun ratkaisun tuli myös olla suorituskyvyltään tehokas. Insinöörityössä keskityttiin erityisesti olemassa olevan projektin tarpeiden huomioimiseen, mahdollisuuteen lisätä uusia tietoturvaskannereita ja järkevästi suunniteltuun luokkarakenteeseen ja toteutukseen. Lisäksi huomiota kiinnitetttiin tietokantamallin toimivuuteen sekä tallennettujen tietojen hyödyllisyyteen.

Työn aikana tuli selväksi niin eri työkalujen kuin raporttienkin rakenteelliset sekä sisällölliset erot. Jokainen soveltuu omaan käyttötarkoitukseensa, mutta tuottaa silti havainnoista kertovan raportin.

Suurimmat haasteet tulivat vastaan tietokantafunktiota ja generaattoritoteutusta suunnitellessa. Tietokantaan liittyvät operaatiot ovat tärkeää toteuttaa oikein, mutta Djangon omaperäiset toimintatavat laittoivat suunnitelmat jatkuvasti uusiksi. Generaattorit taas ovat ennestään tuntematon käsite, mikä aiheutti haasteita niiden käyttämisessä oikealla tavalla. Ongelmat saatiin kuitenkin ratkaistua, ja lopputuloksena syntyi vaatimusten mukaiset jäsentimet, jotka muotoilevat raporttien tulokset yhteiseen rakenteeseen mahdollisimman tehokkaasti.

Jäsentimet toteutettiin kahdella tavalla, joista ensimmäinen oli toteutettu yksinkertaisilla for-silmukoilla, ja toinen Pythonin generaattoreilla. Näiden kykyä suoriutua kolmen raportin jäsentämisestä ja havaintotietojen tallentamisesta mitattiin. Mittaustuloksista selvisi generaattoritoteutuksen olevan nopeampi, jos raportissa on paljon havaintoon liittyviä tunniste- ja referenssitietoja. Keskimäärin toteutusten välillä ei kuitenkaan ollut suuria eroja, mutta generaattoritoteutus valittiin lopulliseksi ratkaisuksi, sillä se toimii joissakin tapauksissa huomattavasti paremmin. Kattavampi testiaineisto varmistaisi oikean valinnan tekemisen, sillä näillä mittauksilla ei tiedetä eri toteutuksien vaikutuksia muihin työkaluihin.

Lähteet

1 FScarfone, Karen; Souppaya, Murugiah; Cody, Amanda; Orebaugh, Angela. Technical Guide to Information Security Testing and Assessment. 2008. Verkkoaineisto. National Institute of Standards and Technology. <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf>. s. 27–28.

2 Muscat, Ian. DAST vs SAST: A Case for Dynamic Application Security Testing. 2019. Verkkoaineisto. Acunetix, Ltd. <https://www.acunetix.com/blog/articles/dast-dynamic-application-security-testing>. Luettu 30.3.2019.

3 Mamaev, Maxim. 2018. Verkkoaineisto. <https://medium.freecodecamp.org/if-you-have-slow-loops-in-python-you-can-fix-it-until-you-cant-3a39e03b6f35>. Luettu 8.3.2019.

4 Anderson, Greg; Neill, Charles; Paz, Jay; Weaver, Aaron. About DefectDojo. 2015. Verkkoaineisto. OWASP Foundation. <https://defectdojo.readthedocs.io/en/latest/models.html>. Luettu 14.3.2019.

5 Weaver, Aaron. DefectDojon etusivu. Verkkoaineisto. OWASP Foundation. <https://www.defectdojo.org/>. Luettu 14.3.2019.

6 Weaver, Aaron. DefectDojon GitHub. Verkkoaineisto. OWASP Foundation. <https://github.com/DefectDojo/django-DefectDojo/tree/master/dojo/tools>. Päivitetty 18.3.2019. Luettu 19.3.2019.

7 Anderson, Greg. Verkkoaineisto. OWASP Foundation. <https://github.com/DefectDojo/django-DefectDojo/blob/master/dojo/models.py#L966>. Päivitetty 20.3.2019. Luettu 30.3.2019.

8 Meet the Metrics: The Code Dx Dashboard. Verkkoaineisto. CodeDx, Inc. <https://codedx.com/dashboard/>. Luettu 4.4.2019.

9 Djangon etusivu. Verkkoaineisto. Django Software Foundation. <https://www.djangoproject.com/>. Luettu 11.3.2019.

10 Why Django? Verkkoaineisto. Django Software Foundation. <https://www.djangoproject.com/start/overview/>. Luettu 11.3.2019.

11 Ni, Min. Web Service Efficiency at Instagram with Python. 2016. Verkkoaineisto. Instagram Engineering. <https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366>. Luettu 16.3.2019.

12 Van Rees, Reinout. Bitbucket, lessons learned - Jesper Noehr. 2011. Verkkoaineisto. <https://reinout.vanrees.org/weblog/2011/06/07/bitbucket.html>. Luettu 16.3.2019.

13 Django Documentation, Release 1.11.18.dev20181203171345. 2018. Verkkoaineisto. Django Software Foundation. <https://media.readthedocs.org/pdf/django/1.11.x/django.pdf>. Luettu 16.3.2019.

14 Windows Example Audit Items. Verkkoaineisto. Tenable, Inc. <https://docs.tenable.com/nessus/compliancechecksreference/Content/WindowsExampleAudits.htm>. Luettu 29.2.2019

15 Rules. Verkkoaineisto. SonarSource SA. <https://docs.sonarqube.org/latest/user-guide/rules/>. Luettu 29.2.2019

16 Category:Vulnerability. 2006. Verkkoaineisto. OWASP Foundation. <https://www.owasp.org/index.php/Category:Vulnerability>. Päivitetty 6.6.2016. Luettu 29.2.2019.

17 Category:Attack. 2006. Verkkoaineisto. OWASP Foundation. <https://www.owasp.org/index.php/Category:Attack>. Päivitetty 6.6.2016. Luettu 29.2.2019.

18 Category:OWASP Top Ten Project. 2006. Verkkoaineisto. OWASP Foundation. <https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project>. Päivitetty 20.10.2017. Luettu 29.2.2019.

19 OWASP Top 10 – 2017, The Ten Most Critical Web Application Security Risks. 2017. Verkkoaineisto. OWASP Foundation. <https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf>. s. 6. Luettu 29.2.2019

20 Security-related Rules. Verkkoaineisto. SonarSource SA. <https://docs.sonarqube.org/latest/user-guide/security-rules/>. Luettu 29.2.2019

21 ZAPping the OWASP Top 10. 2017. Verkkoaineisto. OWASP Foundation. <https://www.owasp.org/index.php/ZAPpingTheTop10>. Päivitetty 26.4.2018. Luettu 29.2.2019.

22 Deraison, Renaud. A Clarification about Nessus Professional. 2017. Verkkoaineisto. Tenable, Inc. <https://www.tenable.com/blog/a-clarification-about-nessus-professional>. Luettu 1.3.2019.

23 Wagner, Jan-Oliver. About Greenbone Community Feed (GCF). 2019. Verkkoaineisto. Greenbone Networks. <https://community.greenbone.net/t/about-greenbone-community-feed-gcf/1224>. Luettu 1.3.2019.

24 OWASP WebGoat Project. 2015. Verkkoaineisto. OWASP Foundation. <https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project>. Päivitetty 3.1.2018. Luettu 1.3.2019.

25 Metasploitable - Virtual Machine to Test Metasploit. 2017. Verkkoaineisto. Rapid7. <https://information.rapid7.com/download-metasploitable-2017.html>. Luettu 1.3.2019.

26 About SonarQube. Verkkoaineisto. SonarSource SA. <https://www.sonarqube.org/about/>. Luettu 23.2.2019.

27 Plans & Pricing. Verkkoaineisto. SonarSource SA. <https://www.sonarsource.com/plans-and-pricing/>. Luettu 23.2.2019.

28 Multi-Language. Verkkoaineisto. SonarSource SA. <https://www.sonarqube.org/features/multi-languages/>. Luettu 23.2.2019

29 SonarQuben Python-kielen haavoittuvuussäännöt. Verkkoaineisto. SonarSource SA. <https://rules.sonarsource.com/python/type/Vulnerability>. Luettu 23.2.2019.

30 SonarQuben Python-kielen tietoturvasäännöt. Verkkoaineisto. SonarSource SA. <https://rules.sonarsource.com/python/type/Security%20Hotspot>. Luettu 23.2.2019.

31 Oftedal, Erlend. What you require you must also retire. Verkkoaineisto. <https://retirejs.github.io/retire.js/>. Luettu 23.2.2019.

32 National Vulnerability Database January 2019. 2019. Verkkoaineisto. National Institute of Standards and Technology. <https://nvd.nist.gov/vuln/full-listing/2019/1>. Luettu 23.2.2019.

33 How does dependency-check work? 2019. Verkkoaineisto. OWASP Foundation. <https://jeremylong.github.io/DependencyCheck/general/internals.html>. Luettu 21.3.2019.

34 OWASP Zed Attack Proxy Project. 2010. Verkkoaineisto. OWASP Foundation. <https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project>. Päivitetty 4.3.2019. Luettu 21.3.2019

35 Kennedy, David; O'Groman, Jim; Kearns, Devon; Aharoni, Mati. 2011. Metasploit - The Penetration Tester's Guide. No Starch Press, Inc. s. 1.

36 Bennetts, Simon. Active Scan. 2015. Verkkoaineisto. OWASP Foundation. <https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsAscan>. Päivitetty 3.6.2015. Luettu 21.3.2019.

37 Bennetts, Simon. Ascanrules. 2015. Verkkoaineisto. OWASP Foundation. <https://github.com/zaproxy/zap-extensions/tree/master/src/org/zaproxy/zap/extension/ascanrules>. Päivitetty 8.10.2018. Luettu 21.3.2019.

38 Bennetts, Simon. Ascanrules. 2015. Verkkoaineisto. OWASP Foundation. <https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsPscan>. Päivitetty 29.3.2017. Luettu 21.3.2019.

39 Carter, Kim. The ZAP API. 2015. Verkkoaineisto. OWASP Foundation. <https://github.com/zaproxy/zaproxy/wiki/ApiDetails>. Päivitetty 31.10.2018. Luettu 21.3.2019.

40 Product Comparison and Sizing. Verkkoaineisto. Greenbone Networks. <https://www.greenbone.net/en/product-comparison/>. Luettu 21.3.2019.

41 Another Perspective: Outside-In Instead of Inside-Out. Verkkoaineisto. Greenbone Networks. <https://www.greenbone.net/en/vulnerability-management/>.

42 OMP: OpenVAS Management Protocol. Verkkoaineisto. Greenbone Networks. <http://www.openvas.org/omp-6-0.html>. Luettu 21.3.2019.

43 Wagner, Jan-Oliver. About GVM Architecture. 2019. Verkkoaineisto. Greenbone Networks. <https://community.greenbone.net/t/about-gvm-architecture/1231>. Luettu 21.3.2019.

44 Retina Network Security Scanner, Unlimited IP Count. Verkkoaineisto. BeyondTrust, Inc. <https://shop.beyondtrust.com/store?SiteID=eeyeinc&Action=DisplayProductDetailsPage&productID=285899100&pgm=94163000&p_d_mrkr=1#Vulnerability-Management>. Luettu 22.3.2019.

45 Free Network Vulnerability Scanner. Verkkoaineisto. BeyondTrust, Inc. <https://www.beyondtrust.com/tools/vulnerability-scanner>. Luettu 22.3.2019.

46 Free Security Tools. Verkkoaineisto. BeyondTrust, inc. <https://www.beyondtrust.com/tools>. Luettu 22.3.2019.

47 Retina Network Security Scanner User Guide. 2015. Verkkoaineisto. BeyondTrust, Inc. <https://www.e-spincorp.com/pdf/product/eEye/Retina-Users-Guide-E-SPIN.pdf>. s. 80. Luettu 22.3.2019.

48 Kushe, R, Phd. 2017. COMPARATIVE STUDY OF VULNERABILITY SCANNING TOOLS: NESSUS vs RETINA. Verkkoaineisto. Faculty of Information Technology – Polytechnic University of Tirana, Albania. <https://stumejournals.com/journals/confsec/2017/2/69/pdf>. s. 69.

49 Mell, Peter; Scarfone, Karen; Romanosky, Sasha. A Complete Guide to the Common Vulnerability Scoring System. 2007. Verkkoaineisto. National Institute of Standards and Technology; Carnegie Mellon University. <https://www.first.org/cvss/v2/guide>. Luettu 2.4.2019.

50 Walding, Ben. Web Service /api/issues. 2014. Verkkoaineisto. SonarSource SA. <https://docs.sonarqube.org/pages/viewpage.action?pageId=2392181>. Luettu 2.4.2019.

51 Bennetts, Simon. ZAP GitHub. Verkkoaineisto. OWASP Foundation. <https://github.com/zaproxy/zaproxy/blob/develop/src/org/parosproxy/paros/core/scanner/Alert.java#L154>. Päivitetty 15.9.2017. Luettu 2.4.2019.

52 Common Vulnerability Scoring System Calculator Version 3. Verkkoaineisto. National Institute of Standards and Technology. <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:A/AC:H/PR:L/UI:R/S:C/C:L/I:L/A:N/CR:H/IR:H/AR:M/MAV:L/MAC:L/MPR:N/MUI:R/MS:C/MC:N/MI:N/MA:L>. Luettu 2.4.2019

53 Parsing. 2003. Verkkoaineisto. Wikimedia Foundation, Inc. <https://en.wikipedia.org/wiki/Parsing>. Päivitetty 1.4.2019. Luettu 5.3.2019.

54 What exactly does "parsing" mean in programming? Verkkoaineisto. Quora. <https://www.quora.com/What-exactly-does-parsing-mean-in-programming>. Luettu 5.3.2019.

55 Python JSON. Verkkoaineisto. Refsnes Data. <https://www.w3schools.com/python/python_json.asp>. Luettu 6.3.2019.

56 Heimes, Christian. Defusedxml GitHub. 2017. Verkkoaineisto. <https://github.com/tiran/defusedxml>. Luettu 6.3.2019.

57 Auto-incrementing primary keys. Verkkoaineisto. Django Software Foundation. <https://docs.djangoproject.com/en/dev/ref/models/instances/#auto-incrementing-primary-keys>. Luettu 7.3.2019.

58 Introduction to Transactions. Verkkoaineisto. Oracle. < https://docs.oracle.com/database/121/CNCPT/transact.htm#CNCPT037>. Luettu 7.3.2019.

59 Khalid, Yasoob. Map, Filter and Reduce. 2017. Verkkoaineisto. <http://book.pythontips.com/en/latest/map_filter.html>. Luettu 8.3.2019.

60 Kruse, Lars. Performance Tips. Verkkoaineisto. Python Software Foundation. <https://wiki.python.org/moin/PythonSpeed/PerformanceTips#Loops>. Luettu 8.3.2019.

61 Szabo, Gabor. List Comprehension vs Generator Expressions in Python. Verkkoaineisto. <https://code-maven.com/list-comprehension-vs-generator-expression>. Luettu 21.4.2019.

62 Python List Comprehensions VS Generator Expressions. 2017. Verkkoaineisto <https://medium.freecodecamp.org/python-list-comprehensions-vs-generator-expressions-cef70ccb49db>. Luettu 8.3.2019.

63 Kern, Robert. line_profiler 2.1.2. 2017. Verkkoaineisto. <https://pypi.org/project/line_profiler/>. Luettu 15.3.2019.

64 Pedregosa, Fabian. memory-profiler 0.55.0. 2018. Verkkoaineisto. 2018.<https://pypi.org/project/memory-profiler/>. Luettu 15.3.2019.

65 Stewart, Alex. Does not work with generator functions? 2013. Verkkoaineisto. <https://github.com/pythonprofilers/memory_profiler/issues/42>. Luettu 15.3.2019.

66 Performing raw SQL queries. Verkkoaineisto. Django Software Foundation. <https://docs.djangoproject.com/en/1.11/topics/db/sql/>. Luettu 26.3.2019.

Liite poistettu julkisesta versiosta salaisena.