**Jami Laamanen**

# Ohjelmistoturvallisuuden parantaminen staattisella analyysillä

Tietotekniikan kandidaatintutkielma

5. toukokuuta 2018

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

**Tekijä:** Jami Laamanen

**Yhteystiedot:** jami.a.laamanen@student.jyu.fi

**Työn nimi:** Ohjelmistoturvallisuuden parantaminen staattisella analyysillä

**Title in English:** Improving software security with static analysis security tools

**Työ:** Kandidaatintutkielma

**Sivumäärä:** 29+0

**Tiivistelmä:** Staattisella analyysillä tarkoitetaan ohjelmiston ominaisuuksien tarkastelemista kääntämättä tai ajamatta ohjelmistoa. Tämä tutkielma on pienimuotoinen kirjallisuuskatsaus, jonka tarkoituksena on lähdekirjallisuutta tutkimalla ja jäsentämällä tutkia, miten staattisella analyysillä voidaan parantaa nykyaikaisten web-applikaatioiden ohjelmistoturvallisuutta. Tutkitun kirjallisuuden perusteella staattista analyysiä hyödyntävillä työkaluilla on mahdollista havaita yleisiä, liiketoiminnan kannalta erittäin tuhoisia haavoittuvuuksia ja täten parantaa merkittävästi ohjelmistoturvallisuutta. Näiden työkalujen tehokas hyödyntäminen ohjelmistokehityksen kontekstissa vaatii kuitenkin suunnitelmallisuutta ja teknistä tietoutta.

**Avainsanat:** Staattinen analyysi, ohjelmistoturvalisuus, ohjelmistokehitys, web-applikaatiot

**Abstract:** Static analysis means examining the properties of software without compiling or executing it. This bachelors thesis is a small-scale literature review that attempts by examining and analysing research literature to explore how static analysis can be used to the improve the security of modern web-applications. Based on the literature examined, static analysis security tools (SAST) are able to detect common software vulnerabilities that can have a disasterous effect on businness, and so are able to significantly improve software security. Using these tools effectively in the context of software development however needs planning and technical knowledge.

**Keywords:** Static analysis, software security, software development, web-applications

# Kuviot

Kuvio 1. Ohjausvuoverkon lähdekoodiesimerkki .......... 4
Kuvio 2. Ohjausvuoverkkokaavioesimerki .......... 4
Kuvio 3. Kutsuvuoverkon lähdekoodiesimerkki .......... 5
Kuvio 4. Kutsuvuoverkkokaavioesimerkki .......... 6
Kuvio 5. SQL-injektion lähdekoodiesimerkki .......... 9
Kuvio 6. XSS-hyökkäyksen lähdekoodiesimerkki .......... 9

# Taulukot

Taulukko 1. Staattista analyysiä hyödyntäviä työkaluja .......... 14

# Sisältö


1 JOHDANTO ........ 1

2 STAATTINEN ANALYYSI ........ 3
2.1 Toimintaperiaate ........ 3
2.2 Dynaamisesti tyypitetyt kielet ........ 6

3 STAATTISTA ANALYYSIÄ HYÖDYNTÄVÄT TYÖKALUT ........ 8
3.1 Ohjelmistohaavoittuvuuksien havaitseminen ........ 8
3.2 Analyysin herkkyydet ........ 10

4 STAATTINEN ANALYYSI OSANA OHJELMISTONKEHITYSPROSESSIA ........ 12
4.1 Työkalut ........ 12
4.2 Integrointi ohjemistokehitykseen ........ 15

5 YHTEENVETO ........ 18

LÄHTEET ........ 20

# 1 Johdanto

Elämme nykyään kasvavissa määrin web-palveluiden keskellä. Tarvitseme niitä moniin päivittäisen elämämme kannalta välttämättömiin toimintoihin, kuten terveydenhoidon palveluiden käyttämiseen, henkilökohtaisen talouden hoitamiseen sekä sosiaalisten suhteiden ylläpitoon. Näiden palveluiden tärkeydestä huolimatta yhdysvaltalaisen tietoturvayhtyö Symantecin (2017) vuosiraportissa vuonna 2017 skannatuista web-applikaatioista 76% sisälsi haavoittuvuuksia. Näistä haavoittuvuuksista 9% arvioitiin kriittisiksi.

Staattisella analyysillä tarkoitetaan ohjelmiston, tai sen osien, tarkastelua ilman ohjelmiston kääntämistä (IEEE 2010, s. 345). Staattista analyysiä hyödyntävillä työkaluilla voidaan automatisoida ja nopeuttaa muutoin aikaa vievää, virheiden löytämiseen tähtäävää manuaalista lähdekoodikatselmustyötä ja täten nopeuttaa myös itse ohjelmistokehitysprosessia. Chess ja McGraw (2004) nostavat artikkelissaan esille, että nämä työkalut vaativat käyttäjiltään merkittävästi vähemmän tietoturvatietoutta ja -taitoa kuin manuaalinen työ mahdollisten ohjelmistohaavoittuvuuksien löytämiseksi. Mitä aikaisemmin haavoittuvuudet, kuten muutkin ohjelmistovirheet huomataan, sitä helpompaa ja kustannustehokkaampaa niiden korjaaminen on (Boehm 1984).

Tehokkaiden työkalujen kehitys on kuitenkin jatkuvaa tasapainottelua analyysin nopeuden sekä haavoittuvuuksien löytämisen tarkkuuden ja luotettavuuden välillä (Gupta, Govil ja Singh 2014) ja voisikin ajatella, että tämän tasapainon löytäminen olisi avainasemassa käytettävyydeltään hyvien työkalujen kehityksessä. Staattisella analyysillä ei myöskään voida mallintaa ohjelmistojen käyttäymistä täydellisesti ja usein työkalut tuottavatkin huomattavia määriä vääriä havaintoja, joka voi pahimmillaan johtaa kehittäjien työmäärän lisääntymiseen (Liu ym. 2012).

Tämän tutkielman tutkimusmetodi on pienimuotoinen kirjallisuuskatsaus Jyväskylän yliopiston informaatioteknologian tiedekunnan tietotekniikan pääaineen kanditutkielmasuosituksen mukaan. Kirjallisuuskatsauksen tarkoitus on lähdekirjallisuu-

teen perehtymisen ja sen jäsentämisen avulla hahmottaa aihepiirin kokonaisuutta, sekä dokumentoida aiheeseen liittyvää tutkimusta ja sen tuloksia.

Tässä tutkielmassa tutkitaan miten staattisella analyysillä voidaan parantaa nykyaikaisten web-applikaatioiden ohjelmistoturvallisuutta. Luvussa 2 tutustutaan staattisen analyysin teoriaan yleisellä tasolla, sen tekniseen toimintaperiaatteeseen sekä dynaamisesti tyypitettyjen ohjelmointikielten esiin tuomiin haasteisiin. Luku 3 tarkastelee staattista analyysiä hyödyntävien työkalujen ominaisuuksia, vahvuuksia ja heikkouksia sekä tutkii miten nämä työkalut hakevat nopeuden ja tarkkuuden välistä tasapainoa. Luvussa 4 vertaillaan eräitä työkaluja ja pohditaan staattisen analyysin integrointia ohjemistokehitysprosessiin ohjelmistoturvallisuuden näkökulmasta.

# 2 Staattinen analyysi

Staattista analyysiä on tutkittu jo 1970-luvulta lähtien (Allen ja Cocke 1976) ja se on pysynyt suosittuna tutkimusaiheena tähän päivään saakka. Uuden suunnan tutkimukselle on antanut staattisen analyysin rooli tietoturvallisuuden parantamisessa (Liu ym. 2012). Tässä kappaleessa perehdytään staattisen analyysin toimintaperiaatteeseen.

## 2.1 Toimintaperiaate

Liu ym. (2012) tiivistävät staattisen analyysin seuraavasti: staattisessa analyysissä luetaan lähdekoodia (tai vaihtoehtoisesti konekieltä) ja etsitään sieltä mahdollisia virheitä, heikkouksia tai haavoittuvuuksia. Staattisen analyysin ajatellaan koostuvan kolmesta eri vaiheesta: parserista, sisäisistä esityksestä sekä sisäisen esityksen analyysistä (Binkley 2007).

Parserin tehtävä on jäsentää lähdekoodia sisäisiksi esityksiksi (Chess ja West 2007, luku 4.1). Binkley (2007) huomauttaa, että vaikka lähdekoodin jäsentäminen ei yleisesti ottaen ole teknisesti vaikeaa, asettavat modernien ohjelmointikielten monimutkaiset ominaisuudet ja esiprosessoidut kielet jäsentämiselle joitakin haasteita ja että jäsennystä voidaan tarvittaessa myös keventää tarkastelemalla vain rajoitettua joukkoa lähdekielen ominaisuuksia.

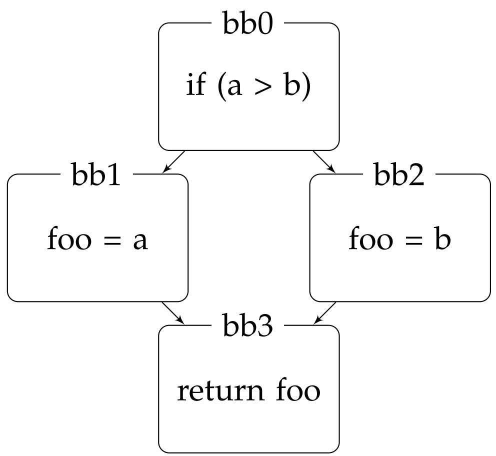
Sisäinen esitys on joukko datarakenteita, jotka kuvaavat lähdekoodia. Niiden tarkoitus on absrahoida lähdekielen ominaisuuksia, jotta ne olisivat helpommin analysoitavissa (Chess ja West 2007, luku 4.1). Binkley (2007) mainitsee ohjausvuoverkon (*control-flow graph*) sekä kutsuvuoverkon (*procedure call graph*) perinteisinä esimerkkeinä sisäisistä esityksistä.

Ohjausvuoverkko kuvaa yksittäisen aliohjelman solmuja, kuten muuttujien arvojen sijoittamista, vertailua sekä ohjausrakenteita (Fischer, Cytron ja LeBlanc 2009, luku 14.2.1). Ohjausvuoverkko rakentuu peruslohkoista. Peruslohko on joukko ko-

Kuvio 1: Ohjausvuoverkon lähdekoodiesimerkki

```
int foo;

if (a > b) foo = a;
   else foo = b;

return foo;
```

Kuvio 2: Ohjausvuoverkkokaavioesimerki




nekielisiä käskyjä, jotka suoritetaan aina alusta loppuun, eikä peruslohkon keskeltä ole mahdollista hypätä ulos. Peruslohkojen avulla voidaan tutkia aliohjelman sisällä olevia suorituspolkuja. Kun ohjelma ajetaan, sen ohjausvuoverkko voidaan esittää suoritettuina peruslohkoina (Chess ja West 2007, luku 4.1). Kaaviossa 2 on esitetty kuvion 1 lähdekoodiesimerkistä rakennettu ohjausvuoverkkokaavio, jossa peruslohkot ovat bb0-bb4.

Kutsuovuoverkko on kuin funktioiden välinen ohjausvuoverkko. Siinä tarkastelun kohteena ovat funktiot, metodit sekä niiden keskinäiset suorituspolut. Fischer, Cytron ja LeBlanc (2009, luku 14.2.1) huomauttavat kutsuvuoverkon muodostamisen olevan yleensä haasteellisempa kuin ohjausvuoverkkojen muodostus. Kutsukaavion avulla voidaan tarkistaa, että ohjelma suoritetaan oikeassa järjestyksessä tai ettei ohjelma sisällä osia, joita ei suoriteta missään tilanteessa. Kuvion 4 kaaviossa

Kuvio 3: Kutsuvuoverkon lähdekoodiesimerkki

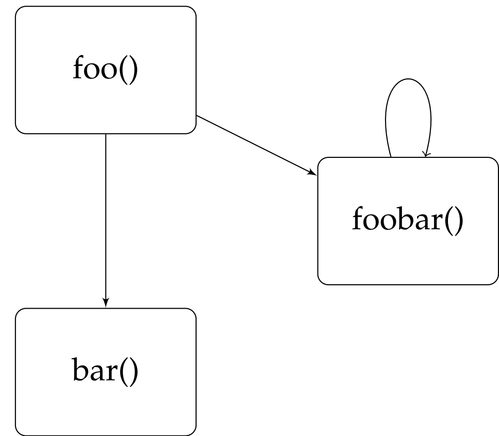
```
public static void foo(int a, int b){
   if (a > b) foobar(a);
      else bar(b);
}

public static void bar(int x){
   System.out.println(x);
}

public static void foobar(int y){
    if (y > 0){
      y--;
      foobar(int y);
   }
}
```

on esitetty kutsuvuoverkkokaavio, joka on muodostettu kuviossa 3 olevan lähdekoodiesimerkin perusteella.

Sekä ohjausvuoverkko, että kutsuvuoverkko ovat esimerkkejä tietovuoanalyysistä (*data flow analysis*), jota voidaan käyttää määrittämään ohjelmiston ajonaikaisia ominaisuuksia (Fischer, Cytron ja LeBlanc 2009, luku 14). Tietovuoanalyysi mahdollistaa ohjelman tutkimisen monelta eri tasolta: lokaali analyysi tarkastelee yksittäistä peruslohkoa ja globaali analyysi (myös *intraprocedual analysis*) tarkastelee aliohjelmia.

Chessin ja Westin (2007, luku 4.2) mukaan edistyneet sisäisten esitysten analyysit ovat yleensä kaksiosaisia. Ne koostuvat globaalista analyysistä sekä aliohjelmien välisestä analyysistä (*interprocedual analysis*). Yhdistämällä intraproseduraalinen sekä interproseduraalinen analyysi saadaan kontekstisidonnaista tietoa ohjelmiston toiminnasta kokonaisuutena.

Kuvio 4: Kutsuvuoverkkokaavioesimerkki




Lopuksi sisäisten esitysten analyysin tulokset esitetään käyttäjälle. Tulokset määrittää säännöstö, joka päättää esimerkiksi, mitkä analyysin tuloksista ovat potentiaalisia haavoittuvuuksia ja kuinka varmoja havainnoista ollaan (Chess ja West 2007, luku 4.3). Säännöstöt voivat myös kuvata toiminnallisuutta, jota ei löydy suoraan lähdekoodista, kuten esimerkiksi ulkoisiten kirjastojen toimintaa.

## 2.2 Dynaamisesti tyypitetyt kielet

Dynaamisesti tyypitetyt kielet ovat suosituimpia kieliä 2010-luvun web-kehityksessä (Hotframeworks 2018) ja Mesbah (2016) nostaakin tutkimuksessaan JavaScriptin suosituimmaksi ohjelmointikieleksi. Syyksi tähän hän esittää JavaScriptin käyttötarkoituksen muuttumista lähinnä web-sivujen dynaamisuutta lisäävien, pienten skriptien kirjoittamisesta kokonaisiin web-applikaatioihin ja palvelinohjelmointiin Node.js-alustan ansiosta. GitHub (2017):n tekemän kyselyn mukaan dynaamisilla ohjelmointikielillä, kuten JavaScript, Python, PHP ja Ruby, oli merkittävä edustus kaikista projekteista.

Dynaamisten kielten erikoisuutena on se, että tullessan virhetilaan ne eivät kaadu, vaan jatkavat suoritusta (Madsen 2015, s. 3). Vaikka tämä toisaalta tekee dynaamisesti tyypitetyistä kielistä sopivia web-ohjelmistojen ohjelmointiin, altistavat nämä pysyvät virhetilat ohjelmistoja haavoittuvuuksille. Näiden mahdollisesti vaikeasti-

kin löydettävien haavoittuvuuksien etsiminen lähdekoodista olisikin erinomainen käyttötarkoitus staattiselle analyysille.

Useat tutkimukset, joissa tutkitaan dynaamisesti tyypitettyjen ohjelmointikielten staattista analyysiä mainitsevat vaikeuksista sovittaa staattinen analyysi tehokkaasti yhteen dynaamisen tyypituksen kanssa tavalla, joka toisi lisäarvoa ohjelmistonkehitysprosessiin. Tällaisia vaikeuksia ovat esimerkiksi JavaScriptin epäintuitiiviset tyypitysominaisuudet (Kashyap ym. 2014), prototyypit ja olioiden luomisen dynamainen luonne (Guarnieri ym. 2011) sekä kirjastojen käytöstä johtuvat ongelmat, kuten jQuery-kirjaston reflektiiviset ominaisuudet (Madsen, Livshits ja Fanning 2013). Tällaiset ongelmat vaikeuttavat staattisen analyysin hyödyntämistä nykyaikaisten web-sovellusten ohjelmistoturvallisuuden parantamisessa.

# 3 Staattista analyysiä hyödyntävät työkalut

Gupta, Govil ja Singh (2014) mainitsevat, että suurin osa ohjelmistohaavoittuvuuksista löytyy lähdekoodista. Tässä kappaleessa käydään läpi miten staattisella analyysillä löydetään haavoittuvuuksia lähdekoodista, millaisia haavoittuvuuksia voidaan havaita sekä minkälaisia seurauksia näihin haavoituvuuksiin kohdistuvilla hyökkäyksillä voi olla. Lisäksi kappaleessa tutustutaan analyysin herkkyyksiin, jotka vaikuttavat työkalujen nopeuteen ja tarkkuuteen.

## 3.1 Ohjelmistohaavoittuvuuksien havaitseminen

SQL-injektiot ja Cross Site Scripting -hyökkäykset ovat esimerkkejä merkittävistä nykyaikaisia web-applikaatioita uhkaavista haavoittuvuuksista. Ohjelmistoturvallisuutta edistävä OWASP-järjestö arvioi vuonna 2017 julkaisemassaan raportissa (OWASP 2017b) injektiot kaikista vaarallisimmiksi web-haavoittuvuuksiksi perustuen niiden yleisyyteen sekä laajoihin haittavaikutuksiin liiketoiminnan kannalta. Samassa raportissa arvioitiin XSS-hyökkäyksien olevan seitsemänneksi vaarallisin haavoittuvuus.

SQL-injektio tapahtuu, kun hyökkääjä saa muutettua SQL-kyselyn käyttäytymistä lisäämällä alkuperäisen kyselyn yhteyteen itse laatimaansa syötteen, joka muuttaa tietokannassa suoritettavan kyselyn toimintaa (Halfond, Viegas, Orso ym. 2006). Tällöin tarkoituksena voi olla tietokannan sisällön kopioiminen tai poistaminen, tai vaikka sisäänkirjautuneen käyttäjän käyttätason nostaminen järjestelmävalvojaksi. Kuviossa 5 esitetyssä tilanteessa syötteellä `1; DROP TABLE *` voitaisiin tuhota koko tietokannan sisältö.

XSS-hyökkäys tapahtuu, kun dynaamisesti luotu web-sivusto näyttää validoimattoman syötteen (Vogt ym. 2007). XSS-hyökkäkset jaetaan yleensä kahteen eri kategoriaan: tallennetut hyökkäykset sekä heijastetut hyökkäykset. Tallennetuissa hyökkäyksissä web-sivustolle tallennetaan vahingollinen skripti, joka ladataan myöhemmin varsinaisen hyökkäyksen tapahtuessa. Heijastetuissa hyökkäyksissä vahingol-

Kuvio 5: SQL-injektion lähdekoodiesimerkki

```
String productId = getParameter("productId");

String query = "SELECT * FROM Products WHERE ProductID = " +
    productId;

ResultSet result = databaseConnection.executeQuery(query);
```

Kuvio 6: XSS-hyökkäyksen lähdekoodiesimerkki

```
<img src="http://evilsite.example/image.jpg">
  <script type="text/javascript">
    var addr =
       "http://evilsite.example/getcoockies.php?cookie=" +
       escape(document.cookie);
  </script>
```

lista skriptiä ei tallenneta, vaan se näytetään hyökkäyksen uhrille dynaamisesti. Tallennetussa hyökkäyksessä voidaan esimerkiksi kuvaan liittää skripti, joka lähettää sisäänkirjautuneen käyttäjän tiedot hyökkääjän sivustolle, kuten kuviossa 6 on tehty.

Yhteistä SQL-injektioille ja XSS-hyökkäyksille niiden vaarallisuuden lisäksi on se, että kummatkin niistä johtuvat syötteiden validoinnin puutteesta. Livshits ja Lam (2005) kategorisoivat SQL-injektiot ja XSS-hyökkäykset osaksi laajempaa haavoittuvuuskategoriaa, koodisaasteita. Tällaisia haavoittuvuuksia voidaan havaita tehokkaasti koodisaasteanalyysillä. Koodisaasteanalyysi[1] (*taint analysis*) on tietovuoanalyysin alalaji (Gupta, Govil ja Singh 2014), joka keskittyy seuraamaan ohjelmaan saapuvia syötteitä tietovirtaa ja tietovuoanalyysiä hyödyntäen (Chess ja West 2007, luku 4.1). Syötteiden vaikutusta ohjelmistossa havainnollistetaan erilaisilla sännöillä.

1. Suomennos on kirjoittajan oma. Ohjelman läpi virtaavat, tarkistamattomat syötteet voivat vaikuttaa tallennettavaan dataan negatiivisesti, *saastuttaen* sen.

Esimerkiksi Livshits ja Lam (2005) määrittelevät yksittäisen haavoittuvuuden saasteongelmaksi, joka koostuu lähdekuvauksesta, jossa syöte tulee ohjelmaan (*source descriptor*), päätepistekuvauksesta, johon validoimattoman syötteen ei tulisi päästä (*sink descriptor*) sekä syötteen kulkemasta polusta (*derivation descriptor*). Sillä Javan tapauksessa merkkijonot ovat muuttumattomia, voidaan potentiaalisia haavoittuvuuksia havaita mikäli jonkin syötteen polku kulkee lähteeltä päätepisteelle. Joissakin ohjelmointikielissä, kuten Rubyssa (Thomas, Fowler ja Hunt 2009, luku 25) ja Perlissä (Wall 2000, luku 20) koodisaasteiden havaitseminen tai sen kaltainen ominaisuus on rakennettu sisälle kieleen.

Guptan, Govilin ja Singhin (2014) mukaan ohjelmistoja kehitetään usein ilman tarvittavaa tietoturvaosaamista tai haavoittuvuuksien korjaamiseen ei kiinnitetä tarpeeksi huomiota. Näin haavoittuuden jäävätkin lähdekoodiin. Vaikka staattisen analyysin hyödyntäminen mahdollistaa lähdekoodissa sijaitsevien ohjelmistohaavoittuvuuksien löytämisen, pätee toisaalta myös päinvastainen väite. Kuten Chess ja West (2007, luku 2.1) toteavat, tulee haavoittuvuuden löytyä lähdekoodista, jotta se voidaan löytää hyödyntämällä staattista analyysiä. Tämä estää mm. ohjelmiston arkkitehtuurin virheistä johtuvat haavoittuvuudet, joihin Santos, Tarrit ja Mirakhorli (2017) luokittelevat esimerkiksi sessionkaappaukset.

## 3.2 Analyysin herkkyydet

Kehitettäessä staattista analyysiä hyödyntäviä työkaluja joudutaan yleensä tasapainottelemaan tarkkuuden ja analyysiin kuluvan ajan välillä. Tätä voidaan hallita määrittämällä analyysin herkkyyksiä. Analyysin herkkyyksillä tarkoitetaan niitä lähdekoodin rakenteellisia asioita, joita analyysi ottaa huomioon. Gupta, Govil ja Singh (2014) esittelevät tutkimuksessaan kolme herkkyyttä: tietovirtaherkkä analyysi (*flow sensitive analysis*), polkuherkkä analyysi (*path sensitive analysis*) ja kontekstiherkkä analyysi (*context sensitive analysis*). Tietovuoanalyysi on luonnostaan tietovirtaherkkää (Nielson, Nielson ja Hankin 2010, luku 2.5.6) ja tyypillsesti polkuvälinpitämätöntä (Huang ym. 2010).

Tietovirtaherkkä analyysi ottaa huomioon suoritettavien lausekkeiden järjestyksen. Se on hitaampaa kuin sen vastakohta, tietovirtavälinpitämätön analyysi, mutta tuottaa tarkempia tuloksia (Nielson, Nielson ja Hankin 2010, s. 101). Lausekkeiden järjestys voidaan selvittää käyttämällä esimerkiksi kontrolliuvokaavioita (Gupta, Govil ja Singh 2014).

Polkuherkkä analyysi ottaa huomioon vain ohjelman suorituksen pätevät polut ja polkuvälinpitämätön analyysi ottaa huomioon kaikki ohjelman suorituksen polut. Polkuvälinpitämätön analyysi vie enemmän aikaa, mutta on tarkempi kuin polkuherkkä analyysi (Gupta, Govil ja Singh 2014).

Kontekstiherkkä analyysi koostuu lokaalista ja globaalista analyysistä (Gupta, Govil ja Singh 2014). Nielson, Nielson ja Hankin (2010, s. 95) esittävät, että kontekstitiedon käyttämisellä saadaan tarkempaa tietoa kutsuttaessa samaa aliohjelmaa useammassa kohdassa ohjelmistoa. Tällöin kahdesta eri kutsusta saatu tieto voidaan erottaa toisistaan. Kontekstiherkkä analyysi on tarkempaa, mutta hitaampaa kuin kontekstivälinpitämätön analyysi.

# 4 Staattinen analyysi osana ohjelmistonkehitysprosessia

Haavoittuvuuksia löytävät työkalut eivät kuitenkaan paranna tietoturvallisuutta yksinään, vaan staattista analyysiä hyödyntäviä työkaluja käyttävät ohjelmistokehittäjät, joilla Chessin ja McGrawin (2004) mukaan on tapana toistaa samoja tietoturvallisuutta vaarantavia virheitä. Tässä kappaleessa pohditaan miten nämä työkalut tulisi integroida ohjelmistonkehitysprosessin, jotta niiden hyöty voitaisiin maksimoida sekä vertaillaan eräitä työkaluja sekä niiden ominaisuuksia.

## 4.1 Työkalut

Taulukossa 1 on esitetty joitakin staattista analyysiä hyödyntävien, vuoden 2010 jälkeen julkaistujen työkalujen vertailua Guptan, Govilin ja Singhin (2014) esittelemien ominaisuuksien perusteella. Työkaluista ilmaiseksi ovat saatavilla LAPSE+ (OWASP 2018) ja Pixy. Maksullisia työkaluja listassa ovat ANDROMEDA, ACTARUS sekä TAJ, jotka toimivat osana IBM:n WALA-kehykstä ja ovat integroituna IBM AppScan-työkaluperheeseen. ACTARUS ja TAJ on kehitetty osana IBM:n Language-based Security (LaBaSec) -hanketta (IBM 2018).

Kaikki vertailtavista työkaluista hyödynsivät koodisaasteanalyysiä ja pystyvät havaitsemaan sekä SQL-injektioita, että XSS-hyökkäyksiä. Koodisaasteanalyysiä voidaan hyödyntää myös muiden injetkiohaavoittuvuuksien havaitsemiseen ja vertailluista työkaluista esimerkiksi LAPSE+ kykenee havaitsemaan komento-, XPath-, XML- ja LDAP-injektioita. Samoja injektioita kykenee havaitsemaan myös IBM:n AppScan (SecToolMarket 2016). Muita työkalujen löytämiä haavoittuvuuksia olivat mm. HTTP-otsikkotietojen muuttaminen (LAPSE+, AppScan) tai ohjelmiston rakenteen paljastavista virheilmoituksista johtuvat hyökkäykset (TAJ). Lisäksi LAPSE+ ja ACTARUS mahdollistivat analyysin helpon laajentamisen. Kaikista työkaluista suppein ominaisuuksiltaan oli Pixy, jonka havaitsemat haavoittuvuuden rajoittuivat vain SQL-injektioihin ja XSS-hyökkäyksiin. Pixyn muutoin nopea ja tarkka ana-

lyysi ei kuitenkaan tue PHP:n oliopohjaisia ominaisuuksia, joka voi johtaa suureen määrään vääriä havaintoja.

Teknisesti katsoen kaikille työkaluille yhteistä oli interproseduraalinen sekä kontekstiherkkä analyysimalli, jota käyttämällä saadaan tarkennettua analyysin lopputulosta. Chess ja West (2007, luku 4.1) havainnollistavat tätä esimerkillä: olisi yksinkertaista raportoida käyttäjälle kaikki haavoittuvaksi arvellun metodin käyttökohdat, mutta ymmärtämällä sitä ympäröivää kontekstia voidaan erottaa mitkä käyttökohdista voivat saada saastuneita syötteitä ja mitkä eivät. Tätä varten tarvitaan funktiorajat ylittävää tietoa ohjelmiston toiminnasta. Analyysin tarkkuutta voidaan parantaa myös muillakin keinoin. Aliasanalyysissä (*alias analysis*) pyritään ymmärtämään mihin muistiosoitteeseen ohjelman muuttuja osoittaa annetulla hetkellä (Appel ja Palsberg 2003, luku 17.5). Ohjelmistoturvallisuuden kannalta on tärkeä tietää voiko esimerkiksi haavoittuvan metodin argumenttina käytetty muuttuja osoittaa toisen muuttujan saastuttamaan muistiosoitteeseen ennen metodin kutsumista.

Tarkka analyysi vaikuttaa kuitenkin työkalun nopeuteen ja vertailtavissa työkaluissa onkin käytetty monia keinoja analyysin nopeuttamiseksi. ANDROMEDA toteuttaa tietovuoanalyysiin tarvittavaa kutsuvuonverkon rakentamista ja aliasanalyysiä laiskasti, vain tarvittaessa. Näin analyysistä saadan nopea uhraamatta tarkkuutta. ACTARUS sen sijaan rakentaa täydellisen kutsuvuoverkon ja aliasanalysoi koko ohjelman. Toiminnan nopeuttamiseksi ACTARUS tutkii kuitenkin vain lähteitä, joihin ylipäätänsä voidaan päästä käsiksi ja lopettaa heti kun syöte validoidaan. Tripp ym. (2013, luku 6) huomauttavat kuitenkin, ettei ACTARUS skaalaudu suurille ohjelmistoille. Rakenteellisten ratkaisujen lisäksi myös käytetyt algoritmit voivat nopeuttaa työkalun toimintaa. LAPSE+ käyttää tietovirtavälinpitämätöntä analyysialgoritmiä (Whaley ja Lam 2004), ACTARUS ja TAJ käyttävät kutsuvuoverkon rakentamisen ja aliasanalyysin yhdistävää, tietovirtavälinpitämätöntä ja tehokasta Andersenin algoritmiä (Andersen 1994), jota on muokattu kontekstiherkäksi sekä Reps-Horwitz-Sagiv -algoritmia, jolla voidaan muuttaa tietovuoverkko-ongelmia reittiongelmiksi ratkaisun nopeuttamiseksi (Reps, Horwitz ja Sagiv 1995).

Erikoisuutena TAJ yhdistää tietovirtavälinpitämätöntä päättelyä keon tilasta ja tietovirta- ja kontekstiherkkää päättelyä ohjelman muuttujista hybridiviipaloinniksi (*hybrid thin slicing*). Viipaloinnilla tarkoitetaan tekniikkaa, jossa suuresta ohjelmasta pyritään tunnistamaan tutkittavan lausekkeen tai arvon kannalta kiinnostava osa (Sridharan, Fink ja Bodik 2007). Viipalointi soveltuu hyvin koodisaasteanalyysiin ja sillä saavutetaan hyvä tasapaino tarkkuuden ja nopeuden välillä. Lisäksi TAJ nopeuttaa analyysiä mm. optimoimalla tarkisteltavaa lähdekoodia, appoksimoimalla kehityskehyksien käyttäytymistä ja priorisoimalla kutsukaavion rakentamista mikäli sille annetut muisti- tai aikarajoitteet ylitetään.

Taulukko 1: Staattista analyysiä hyödyntäviä työkaluja

| Työkalu | Tutkimus | Analyysitekniikat | Analysoitava kieli |
|---|---|---|---|
| ANDROMEDA | (Tripp ym. 2013) | Laiska kontekstiherkkä<br>+ Interproseduraalinen<br>+ Laiska alias | Java<br>+ .NET<br>+ JavaScript |
| ACTARUS | (Guarnieri ym. 2011) | Kontekstiherkkä<br>+ Interproseduraalinen<br>+ Alias | JavaScript |
| LAPSE+ | (Pérez, Filipiak ja Sierra 2011) | Kontekstiherkkä<br>+ Interproseduraalinen<br>+ Tietovirtavahva<br>+ Alias | Java |
| Pixy | (Jovanovic, Kruegel ja Kirda 2010) | Tietovirtaherkkä<br>+ Interproseduraalinen<br>+ Kontekstiherkkä<br>+ Alias | PHP |
| TAJ | (Tripp ym. 2009) | Hybridiviipalointi<br>+ Kontekstiherkkä +<br>+ Interproseduraalinen<br>+ Alias | Java |

## 4.2 Integrointi ohjemistokehitykseen

Perinteisesti ohjelmistonkehityksessä haavoittuvuuksia ja muita vikoja etsitään ja korjataan ohjelmiston testausvaiheessa, jolloin merkittävä osa ohjelmiston komponenteista on jo kirjoitettu. Staattisen analyysin avulla ohjelmistohaavoittuvuuksien löytäminen ja korjaaminen tapahtuu jo ohjelmiston kehitysvaiheessa, eikä testausvaiheessa. Näin puututaankin Chessin ja Westin (2007, luku 1.4) mukaan haavoittuvuuksien syihin, eikä oireisiin. Staattista analyysiä on mahdollista hyödyntää useassa eri tilanteessa ohjelmistokehityksen kehitysvaiheessa (Johnson ym. 2013). Näitä ovat esimerkiksi kehittäjän tehdessä analyysin lähdekoodista halutessaan, sovitun ajanjakson välein (vaikkapa päivittäin), tai aina kun lähdekoodia halutaan lisätä kehitystiimin versionhallintaan.

Staattisen analyysin hyötyinä pidetään ohjelmiston tarkastamisen nopeutta ja laajuutta verrattuna manuaaliseen lähdekoodikatselmukseen. Tarkastelussa on koko lähdekoodi ilman kehittäjien mahdollisia ennakkoluuloja siitä, mikä osa lähdekoodista on kiinnostavaa tarkastelun kannalta (Chess ja West 2007, luku 2.1). Liu ym. (2012) huomauttavat kuitenkin, että staattisella analyysillä voidaan vain arvioida ohjelmiston ajonaikaista käyttäytymistä, eikä tämä arvio voi koskaan olla täydellinen mallinnus ohjelmistosta. Näin ollen työkalut joutuvat tekemään approksimaatioita ohjelmiston toiminnasta, joka johtaa lopputulosten mahdolliseen vääristymiseen. Tämä ilmenee väärinä havaintoina, jotka vaativat ohjelmistokehittäjän työtä havaintojen varmistamiseen ja korjaamiseen. (Chess ja West 2007, luku 1.4). Staattista analyysiä hyödyntävillä työkaluilla ei voidakaan automatisoida koko lähdekoodikatselmusprosessia. Chess ja West (2007, luku 3.1) suosittelevat integroimaan staattisen analyysin koodikatselmusprosessiin, jossa lähdekoodi analysoidaan, analyysin tulokset katselmoidaan ja virheet korjataan.

Johnson ym. (2013) tekemän tutkimuksen mukaan ohjelmistokehittäjille on tärkeää oli tulosten selkeä esittäminen, työkalujen sopiminen kehitysprosessiin sekä työkalujen laajennusmahdollisuudet. Tutkimuksessa haastatellut kehitäjät kokivat käyttämissään työkaluissa enemmän negatiivisia kuin positiivisia puolia. Tulevaisuudessa työkalujen tulisikin tarjota entistä enemmän tietoa löydetyistä haavoittuvuuk-

sista ja niiden korjauksesta tukeakseen kehittää, sekä integroitua paremmin sekä yksittäisen kehittäjän, että koko kehitystiimin kehitystyöhön. Kehittäjän työtä haittaavat ominaisuudet voivatkin vaikeuttaa tai jopa estää staattisen analyysin käyttöä. Käytettävyyden lisäksi tärkeää oikean työkalun valinnassa Boote (2016) esittää olevan tärkeää projektin ohjelmointikielen tuki, työkalun tarkoitettu käyttöryhmä, analyysin käyttötarkoitus sekä lisenssin hinnan. Jotkin työkalut tarjoavat suurille yrityksille tarkoitettuja toimintoja, joita pieni kehitystiimi ei tarvitse ja jotkin työkalut eivät välttämättä tue esimerkiksi CI-integraatiota. OWASP (2017a) mainitsee tärkeiksi kriteereiksi lisäksi integraation käytettävään ohjelmointiympäristöön sekä työkalun havaitsemat haavoittuvuudet.

Eräs keino hallita suurta määrää vääriä havaintoja on staattisen analyysin yhdistäminen dynaamiseen analyysiin, joka kerää tietoja ohjelmistosta ajon aikana. Zhang ym. (2011) esittävät staattisen analyysin vahvuudeksi käydä läpi suuria määriä lähdekoodia huonolla tarkkuudella ja dynaamisen analyysin vahvuudeksi päinvastaisesti käydä läpi pieniä osia lähdekoodia tarkasti. Heidän kehittämänsä SDCF-kehys seuraa dynaamisesti saastuneeksi merkattuja syötteitä analysoiden samalla staattisesti syötteen mahdollista käyttäytymistä ohjelmistossa. Balzarottin ym. (2008) kehittämä SANER taas analysoi staattisesti syötteiden vaikutuksia ohjelmstossa ja tarkistaa syötekenttien sanitaation toimintaa dynaamisesti.

Yksittäisten staattista analyysiä hyödyntävien työkalujen huomaamien haavoittuvuuksien ja virheiden määrässä on myös eroja, eivätkä nämä työkalut huomaa kaikkia mahdollisia ongelmia. Okun ym. (2007) tekemässä tutkimuksessa yksittäiset edistyneet työkalut löysivät keskimäärin puolet ohjelmistojen haavoittuvuuksista. Joissain tutkimuksissa ratkaisuksi tähän on esitetty useampien työkalujen ja metodien yhdistämistä. Nunes ym. (2017) totesivat, että useiden työkalujen yhdistäminen johtaa useampien haavoittuvuuksien löytämiseen, vaikkakin kaikkia haavoittuvuuksia ei siltikään välttämättä löydetä. Suurempi määrä eri työkaluja johti suurempaan määrään vääriä havaintoja. Toisaalta Goseva-Popstojanova ja Perhinschi (2015) totesivat testatessaan kolmea työkalua, etteivät modernit työkalut ole kovinkaan tehokkaita löytämään ohjelmistohaavoittuvuuksia. Useiden työkalujen rinnakkai-

nen käyttö edellyttää kehitystiimiltä siis jatkuvaa tasapainottelua haavoittuvuuksien löytämisen tarkkuuden ja väärien havantojen välillä. 41% 122:sta suosituimmasta vapaan lähdekoodin projektista hyödynsi staattista analyysiä ohjelmistovirheiden etsinnässä. Lisäksi 22% näistä projekteista käytti virheiden etsintään kahta työkalua, ja 14% kolmea (Beller ym. 2016).

# 5 Yhteenveto

Staatisella analyysillä on mahdollista parantaa ohjelmiston tietoturvallisuutta merkittävästi ja sen avulla voidaankin havaita liiketoiminnan kannalta tuhoisasti vaikuttavia haavoittuvuuksia, kuten SQL-injektioita ja XSS-hyökkäyksiä. Nämä haavoittuvuudet voivat OWASP:n (2017b) mukaan johtaa tietokantojen sisällön tai käyttäjätietojen vuotamiseen tai koko tietokannan sisällön menettämiseen. Lähdekoodista voidaan havaita muitakin haavoittuvuuksia, kuten muita injektioita. Nykyaikaisilla, staattista analyysiä hyödyntävillä työkaluilla voidaan saavuttaa laaja lähdekoodin analysoinnin kattavuus nopeasti ja työkalut, kuten ANDROMEDA ja TAJ, skaalautuvat hyvin suurillekin ohjelmistoille erinäisten teknisten ratkaisujen ansiosta.

Vaikka analyysi on nopeaa ja kattavaa, jättävät sen tarkkuus ja varmuus usein varaa parantamiselle. Tämä johtaakin usein suureen määrään epävarmoja havaintoja, joista osa on vääriä. Staattinen analyysi vaatii manuaalista työtä tulosten katselmointiin ja haavoittuvuuksien korjaamiseen. Kaikkien haavoittuvuuksien havaitseminen ei myöskään ole mahdollista, sillä osaa niistä, kuten sessionkaappauksia tai kryptografisia ongelmia, ei voi havaita lähdekoodista. Näin ollen staattisen analyysin käyttö ei ole kokonaisvaltainen ratkaisu tietoturvan parantamiseen. Myös työkalujen käytettävyydessä on usein parannettavaa.

Parhaimmillaan staattinen analyysi tarjoaa koko lähdekoodipohjan kattavan, alustavan kartoituksen ohjelmiston tietoturvasta vaatimatta kehittäjiltä merkittävää tietoturvatietoutta. Tämä kartoitus toimii hyvnä pohjana kehitystiimin koodikatselmukselle, jossa käydään läpi analyysin tulokset ja yksittäisten havaintojen varmuus ja vaikuttavuus sekä päätetään korjaustoimenpiteistä tulosten pohjalta. Lisäksi haavoittuvuudet löydetään jo ohjelmointivaiheessa, jolloin niiden korjaus on yleensä helpompaa ja kustannustehokkaampaa. Pahimmillaan taas työkalujen käyttöönotto johtaa kehitystiimin ylimääräiseen kuormittamiseen suurella havaintomäärällä ja työkaluilla, jotka eivät sovellu käyttötarkoitukseen. Tämän kaltaisilta suurilta ongelmilta säästytään asettamalla tietoturvan kehittämiselle tarkkoja tavoitteita

(Chess ja West 2007, luku 3.1), valitsemalla käytettävät työkalut huolella ja varaamalla tarpeeksi aikaa niiden asentamiseen, konfigurointiin ja käyttöönottoon.

# Lähteet

Allen, F. E., ja J. Cocke. 1976. "A Program Data Flow Analysis Procedure". *Commun. ACM* (New York, NY, USA) 19, numero 3 (maaliskuu): 137–. ISSN: 0001-0782. doi:10.1145/360018.360025.

Andersen, Lars Ole. 1994. "Program analysis and specialization for the C programming language". Tohtorinväitöskirja, Department of Computer Science, University of Copenhagen.

Appel, Andrew W., ja Jens Palsberg. 2003. *Modern Compiler Implementation in Java.* 2nd. New York, NY, USA: Cambridge University Press. ISBN: 052182060X.

Balzarotti, Davide, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel ja Giovanni Vigna. 2008. "Saner: Composing static and dynamic analysis to validate sanitization in web applications". Teoksessa *Security and Privacy, 2008. SP 2008. IEEE Symposium on,* 387–401. IEEE.

Beller, Moritz, Radjino Bholanath, Shane McIntosh ja Andy Zaidman. 2016. "Analyzing the state of static analysis: A large-scale evaluation in open source software". Teoksessa *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on,* 1:470–481. IEEE.

Binkley, David. 2007. "Source Code Analysis: A Road Map". Teoksessa *2007 Future of Software Engineering,* 104–119. FOSE '07. Washington, DC, USA: IEEE Computer Society. ISBN: 0-7695-2829-5. doi:10.1109/FOSE.2007.27.

Boehm, B. W. 1984. "Software Engineering Economics". *IEEE Transactions on Software Engineering* SE-10, numero 1 (tammikuu): 4–21. ISSN: 0098-5589. doi:10.1109/TSE.1984.5010193.

Boote, J. 2016. "5 questions to ask yourself when deciding on the best static code analysis tool". Viitattu 7. huhtikuuta 2018. https://www.synopsys.com/blogs/software-security/questions-best-static-code-analysis-tool/.

Chess, B., ja G. McGraw. 2004. "Static analysis for security". *IEEE Security Privacy* 2, numero 6 (marraskuu): 76–79. ISSN: 1540-7993. doi:10.1109/MSP.2004.111.

Chess, Brian, ja Jacob West. 2007. *Secure Programming with Static Analysis.* First. Addison-Wesley Professional. ISBN: 9780321424778.

Fischer, Charles N., Ronald K. Cytron ja Richard J. LeBlanc. 2009. *Crafting A Compiler.* 1st. USA: Addison-Wesley Publishing Company. ISBN: 0136067050, 9780136067054.

GitHub. 2017. "Developer Survey Results". Viitattu 17. helmikuuta 2018. https://insights.stackoverflow.com/survey/2017.

Goseva-Popstojanova, Katerina, ja Andrei Perhinschi. 2015. "On the capability of static code analysis to detect security vulnerabilities". *Information and Software Technology* 68:18–33.

Guarnieri, Salvatore, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet ja Ryan Berg. 2011. "Saving the World Wide Web from Vulnerable JavaScript". Teoksessa *Proceedings of the 2011 International Symposium on Software Testing and Analysis,* 177–187. ISSTA '11. Toronto, Ontario, Canada: ACM. ISBN: 978-1-4503-0562-4. doi:10.1145/2001420.2001442.

Gupta, M. K., M. C. Govil ja G. Singh. 2014. "Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey". Teoksessa *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014),* 1–5. Toukokuu. doi:10.1109/ICRAIE.2014.6909173.

Halfond, William G, Jeremy Viegas, Alessandro Orso ym. 2006. "A classification of SQL-injection attacks and countermeasures". Teoksessa *Proceedings of the IEEE International Symposium on Secure Software Engineering,* 1:13–15. IEEE.

Hotframeworks. 2018. "Web framework rankings". Viitattu 20. helmikuuta 2018. https://hotframeworks.com/.

Huang, Jianjun, Bin Liang, Jiagui Zhong, Qianqian Wang ja Jingjing Cai. 2010. "Vulnerabilities static detection for Web applications with false positive suppression". Teoksessa *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on*, 574–577. IEEE.

IBM. 2018. "LaBaSec: Language-based Security". Viitattu 25. maaliskuuta 2018. https://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1599.

IEEE. 2010. "ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary". *ISO/IEC/IEEE 24765:2010(E)* (joulukuu): 1–418. doi:10.1109/IEEESTD.2010.5733835.

Johnson, Brittany, Yoonki Song, Emerson Murphy-Hill ja Robert Bowdidge. 2013. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" Teoksessa *Proceedings of the 2013 International Conference on Software Engineering*, 672–681. ICSE '13. San Francisco, CA, USA: IEEE Press. ISBN: 978-1-4673-3076-3.

Jovanovic, Nenad, Christopher Kruegel ja Engin Kirda. 2010. "Static analysis for detecting taint-style vulnerabilities in web applications". *Journal of Computer Security* 18 (5): 861–907.

Kashyap, Vineeth, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann ja Ben Hardekopf. 2014. "JSAI: A Static Analysis Platform for JavaScript". Teoksessa *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 121–132. FSE 2014. Hong Kong, China: ACM. ISBN: 978-1-4503-3056-5. doi:10.1145/2635868.2635904.

Liu, B., L. Shi, Z. Cai ja M. Li. 2012. "Software Vulnerability Discovery Techniques: A Survey". Teoksessa *2012 Fourth International Conference on Multimedia Information Networking and Security*, 152–156. Marraskuu. doi:10.1109/MINES.2012.202.

Livshits, V Benjamin, ja Monica S Lam. 2005. "Finding Security Vulnerabilities in Java Applications with Static Analysis." Teoksessa *USENIX Security Symposium*, 14:18–18.

Madsen, Magnus. 2015. “Static Analysis of Dynamic Languages”. Tohtorinväitöskirja, Department of Computer Science, Aarhus University.

Madsen, Magnus, Benjamin Livshits ja Michael Fanning. 2013. “Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries”. Teoksessa *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering,* 499–509. ESEC/FSE 2013. Saint Petersburg, Russia: ACM. ISBN: 978-1-4503-2237-9. doi:10.1145/2491411.2491417.

Mesbah, A. 2016. “Software Analysis for the Web: Achievements and Prospects”. Teoksessa *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER),* 5:91–103. Maaliskuu. doi:10.1109/SANER.2016.109.

Nielson, Flemming, Hanne R. Nielson ja Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated. ISBN: 3642084745, 9783642084744.

Nunes, Paulo, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia ja Marco Vieira. 2017. “On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study”. Teoksessa *2017 13th European Dependable Computing Conference (EDCC),* 121–128. IEEE.

Okun, Vadim, William F. Guthrie, Romain Gaucher ja Paul E. Black. 2007. “Effect of Static Analysis Tools on Software Security: Preliminary Investigation”. Teoksessa *Proceedings of the 2007 ACM Workshop on Quality of Protection,* 1–5. QoP ’07. Alexandria, Virginia, USA: ACM. ISBN: 978-1-59593-885-5. doi:10.1145/1314257.1314260.

OWASP. 2017a. “Static Code Analysis”. Viitattu 7. huhtikuuta 2018. https://www.owasp.org/index.php/Source_Code_Analysis_Tools.

———. 2017b. “Top 10-2017 Top 10”. Viitattu 27. tammikuuta 2018. http://www.owasp.org/index.php/Top_10-2017_Top_10.

———. 2018. “OWASP LAPSE Project”. Viitattu 25. maaliskuuta 2018. http://www.owasp.org/index.php/OWASP_LAPSE_PROJECT.

Pérez, Pablo Martín, Joanna Filipiak ja José María Sierra. 2011. "LAPSE+ Static Analysis Security Software: Vulnerabilities Detection in Java EE Applications". Teoksessa *Future Information Technology,* toimittanut James J. Park, Laurence T. Yang ja Changhoon Lee, 148–156. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-22333-4.

Reps, Thomas, Susan Horwitz ja Mooly Sagiv. 1995. "Precise interprocedural dataflow analysis via graph reachability". Teoksessa *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* 49–61. ACM.

Santos, Joanna CS, Katy Tarrit ja Mehdi Mirakhorli. 2017. "A Catalog of Security Architecture Weaknesses". Teoksessa *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on,* 220–223. IEEE.

SecToolMarket. 2016. "Detailed Web Application Scanner Information - IBM AppScan - WAVSEP Benchmark 2014/2016". Viitattu 25. maaliskuuta 2018. `www.sectoolmarket.com/web-application-scanners/76.html`.

Sridharan, Manu, Stephen J Fink ja Rastislav Bodik. 2007. "Thin slicing". *ACM SIGPLAN Notices* 42 (6): 112–122.

Symantec. 2017. "Internet Security Threat Report, vol. 22". Viitattu 27. tammikuuta 2018. `http://www.symantec.com/security-center/threat-report`.

Thomas, Dave, Chad Fowler ja Andy Hunt. 2009. *Programming Ruby 1.9: The Pragmatic Programmers' Guide.* 3rd. Pragmatic Bookshelf. ISBN: 1934356085, 9781934356081.

Tripp, Omer, Marco Pistoia, Patrick Cousot, Radhia Cousot ja Salvatore Guarnieri. 2013. "Andromeda: Accurate and Scalable Security Analysis of Web Applications". Teoksessa *Fundamental Approaches to Software Engineering,* toimittanut Vittorio Cortellessa ja Dániel Varró, 210–225. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-37057-1.

Tripp, Omer, Marco Pistoia, Stephen J Fink, Manu Sridharan ja Omri Weisman. 2009. "TAJ: effective taint analysis of web applications". Teoksessa *ACM Sigplan Notices,* 44:87–97. 6. ACM.

Wall, Larry. 2000. *Programming Perl.* 3rd. Toimittanut Mike Loukides. Sebastopol, CA, USA: O'Reilly & Associates, Inc. ISBN: 0596000278.

Whaley, John, ja Monica S Lam. 2004. "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams". Teoksessa *ACM SIGPLAN Notices,* 39:131–144. 6. ACM.

Vogt, Philipp, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel ja Giovanni Vigna. 2007. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis." Teoksessa *NDSS,* 2007:12.

Zhang, Ruoyu, Shiqiu Huang, Zhengwei Qi ja Haibin Guan. 2011. "Combining static and dynamic analysis to discover software vulnerabilities". Teoksessa *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on,* 175–181. IEEE.