

SQL-Injektio ja siltä suojautuminen

Lalli Nuorteva

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 2. maaliskuuta 2015

Sisältö

1	Johdanto	1
2	Käsitteitä	1
3	SQL-Injektio	2
3.1	SQL-Injektio käytännössä	2
3.1.1	Obfuskointi	3
3.1.2	SQL-Injektion alatyypit	3
4	Tietoturvallisen ohjelman toteutus	4
4.1	Parametrisoidut kyselyt	4
4.2	Korvaaminen ja validointi	4
4.3	Datan validointi	4
4.4	Arkaluontoisen datan säilytys	5
4.5	Hienojakoinen pääsynhallinta tietokantaan	5
4.5.1	Istuntoon perustuva parametrisointi metodi	5
4.6	AMNESIA	6
5	Ohjelmiston testaus SQL-injektioden varalta	8
5.1	Musta laatikko -testaus	8
5.1.1	Syötteiden luominen	9
5.2	Valkolaatikkotestaus	9
6	Yhteenveto	10
	Lähteet	10

1 Johdanto

Tämän kandidaatintutkielman päätavoite on selvittää kuinka SQL-injektiolta voidaan suojautua mahdollisimman tehokkaasti. Tutkielmani jakautuu kolmeen pääaiheeseen. Ensimmäisissä luvuissa esitellään mikä on SQL-injektio. SQL-Injektio esitellään ensin pääpiirteissään. Sen jälkeen esitellään miten se voidaan toteuttaa käytännössä, sekä millaisiin eri alatyyppeihin SQL-injektiot jaotellaan. Toisessa kappaleessa tutkitaan erilaisia menetelmiä ja ohjelmointikäytänteitä joilla SQL-injektioita voidaan ehkäistä. Kolmannessa kappaleessa keskitytään sovelluksen penetraatiotestaamisen erilaisilla menetelmillä.

Tutkimusten mukaan 8% web-palveluista sisältää haavoittuvuuksia. 87% Näistä haavoittuvuuksista on SQL-injektio haavoittuvuuksia [10]. Yleisyytensä lisäksi SQL-injektio on myös hyvin vaarallinen. Onnistuneen SQL-injektion avulla hyökkääjä voi suorittaa huonosti suojatussa tietokannassa mitä tahansa operaatioita. Tämä mahdollistaa esimerkiksi arkuluontoisten tietojen lukemisen ja muokkaamisen, tai esimerkiksi sovelluksen autentikaation ohittamisen. SQL-Injektiolla on vuosien aikana tehty lukuisia murtoja. SQL-Injektio mielletään monesti amatöörien ongelmaksi, mutta sen avulla on murrettu myös paljon ammattilaisten ohjelmoimia järjestelmiä. Yksi suurimmista SQL-Injektion avulla tehdyistä murroista tehtiin Guess.com:ille. Hyökkääjä sai käsinsä 200 000 ihmisen nimet ja luottokorttitiedot.

Sen lisäksi että SQL-injektio on yleisin tietoturva-aukko, se on myös helppoa toteuttaa ilman suurempaa ymmärrystä sen toiminnasta. SQL-Injektioiden tekemiseen löytyy valmiita työkaluja, jotka ilmoittavat sivuston heikkouksista lähes tulkoon napin painalluksella. Tästä esimerkkinä "sqlmap"-työkalu joka on tehty penetraatiotestaamista varten (*engl. penetration testing*).

Erilaisten palveluiden siirtyessä verkkoon, verkossa käsitellään jatkuvasti entistä enemmän arkaluontoisia tietoja. Verkossa hoidetaan asioita kuten laskujen maksaminen, hotellien varaus ja henkilökohtaisten viestien vaihtaminen. Yleensä tiedot tallennetaan relaatiotietokantoihin. Juuri tällaiset sovellukset voivat olla haavoittuvaisia SQL-Injektioille, ellei asiaa ole otettu huomioon. Tästä syystä jokaisen tietokantasovelluksia ohjelmoivan on ymmärrettävä mikä on SQL-injektio ja kuinka suojautua siltä voidaan suojautua.

2 Käsitteitä

-SQL - SQL Injection out of band inband

3 SQL-Injektio

Anleyn artikkelin "Advancen SQL Injections in SQL Server Applications" mukaan SQL-Injektio hyökkäys esiintyy silloin, kun hyökkääjä pääsee muuttamaan SQL käskyn logiikkaa, semantiikkaa tai syntaksia. Tämä tapahtuu lisäämällä alkuperäiseen kyselyyn uusia SQL avainsanoja tai operaattoreita. Mikäli sovelluksen tietokantaoikeuksia ei ole erikseen rajattu, hyökkääjä voi onnistuneen SQL-injektion seurauksena suorittaa mitä tahansa tietokantapalvelimen tukemia SQL-kyselyitä. Jotkut tietokantapalvelimet sallivat myös käyttöjärjestelmätason kommentojen suorittamisen. Tällöin hyökkääjän on mahdollista suorittaa myös muunlaisia hyökkäyksiä.

SQL-Injektio on mahdollinen vain silloin kun käyttäjältä tulevaa tietoa käytetään osana tietokantapalvelimelle tehtävää kyselyä. Tämä on kuitenkin varsin tavallinen tarve sovelluksissa. Tietokannassa voidaan säilyttää esimerkiksi käyttäjätunnuksia ja salasanoja. Näin ollen kirjautuessa järjestelmään käyttäjän antamaa syötettä käytetään osana SQL-kyselyä.

3.1 SQL-Injektio käytännössä

Esimerkiksi tuotteiden etsimiseen liittyvä SQL-käsky voidaan rakentaa huonosti suunnitellussa sovelluksessa seuraavalla tavalla:

```
sql = "SELECT * FROM tuotteet  
WHERE nimi =" + params[:tuotenimi]
```

Mikäli käyttäjä antaa nimekseen "; DROP TABLE tuotteet". Valmis kysely tietokannalle näyttää seuraavalta:

```
sql = "SELECT * FROM tuotteet  
WHERE nimi = ''; DROP TABLE tuotteet"
```

Kyseinen kysely etsii ensin kaikki tuotteet joiden nimi on tyhjä. Seuraavaksi suoritetaan komento "DROP TABLE tuotteet", joka poistaa koko tuotteet taulun.

"DROP TABLE tuotteet" tilalla olisi voinut olla mikä tahansa muukin SQL-kysely. Esimerkiksi kaikkien tuotteiden listaamiseen hyökkääjä olisi voinut käyttää tuotenimeä joka sisältää jonkin tautologian esimerkiksi "; OR 1=1". Liittääkseen vastaukseen jonkin muun taulun hyökkääjä olisi voinut käyttää UNION:ia.

3.1.1 Obfuskointi

Salgadon kirjoittaman "SQL Injection Optimization and Obfuscation Techniques"[7] artikkelin mukaan erilaisia palomuuureja voidaan yrittää kiertää obfuskoinnilla. Yksinkertaisimmillaan obfuskointi voi olla esimerkiksi "DROP" avainsanan muuttaminen "DroP":iksi.

Monet hienostuneemmat tavat käyttävät SQL-injektioiden piilottamiseen esimerkiksi erilaisia enkoodauksia. Salgagon mukaan enkoodauksien käyttö perustuu siihen, että eri kerrokset käsittelevät enkoodauksia eri tavalla. Esimerkiksi Unicodessa merkkiä "a" vastaa merkkijono "%u0061". Voi olla että palomuuuri tulkitsee merkkijonon "%u0061" tavallisena merkkijonona, kun taas tietokanta tulkitsee sen kirjaimena "a". Näin ollen esimerkiksi avainsana SELECT voidaan piilottaa unicoden avulla merkkijonoon "%u0053%u0045%u004c%u0045%u0043%u0054".

3.1.2 SQL-Injektion alatyypit

Artikkelin "SQL-Injection is still alive" mukaan SQL-injektioita on kolmea eri päätyyppiä[6].

Inband injektio

Kun SQL-injektion tuloste saadaan samaa reittiä kun se on syötetty, on kyseessä inband injektio. Esimerkiksi jos sovelluksessa on mahdollista hakea lista tuotteista jotka ovat maasta jonka käyttäjä antaa kyselyssä.

Out-of-band injektio

Kun SQL-Injektion tuloste saadaan eri reittiä kun se on syötetty, on kyseessä out-of-band injektio. Web-sovellus saattaa esimerkiksi tallettaa tietokantaansa millä selaimilla sitä on käytetty. Selaintiedot haetaan HTTP pyynnön "User-Agent"-kentästä. Hyökkääjä voi asettaa SQL-injektion User-Agent kenttäänsä. Todennäköisesti hyökkääjä ei näe kyselyn tulosta kyselyn palauttamalla sivulla. Hyökkääjän voi ohjata tulokset itselleen esimerkiksi suorittamalla injektiossa haun:

```
utl_http('http://www.hyokkaajansivu.fi/
injections/' ||
SELECT password
FROM User
WHERE username = 'admin'
)
```

Injektion onnistuessa hyökkääjän palvelimen logeissa näkyy esimerkiksi:

```
GET "/injections/admininpassword", 200
```

Tällöin hyökkääjä saa selville käyttäjän admin salasanan.

Sokea injektio

Sokea injektio: Hyökkääjä ei saa minkäänlaista palautetta sovellukselta. Hyökkääjä voi kuitenkin kokeilla muokata sovelluksen tietoja ja tarkastella vaikuttaako se sovellukseen. Hyökkääjä voi käyttää apunaan sitä kuinka nopeasti sivu latautuu. Lisäämällä injektoituun sql-käskyyn komennon "waitfor delay 0:0:5", tietokanta odottaa 5 sekuntia ennen kuin se palauttaa tuloksen. Tästä voidaan päätellä injektion onnistuneen. [8]

4 Tietoturvallisen ohjelman toteutus

SQL-Injektioilta suojautumiseen on kehitetty useita keinoja. *TÄHÄN TII-VISTELMÄÄ ALLAOLEVISTA YMS*

4.1 Parametrisoidut kyselyt

Parametrisoiduissa kyselyssä luodaan SQL-kyselystä pohja, johon lisätään paikanpitäjät (*engl. placeholder*). Parametrisoitu kysely annetaan tietokannalle. Tietokanta kääntää ja optimoi kyselyn pohjan vain kerran. Tietokanta ei kuitenkaan vielä suorita varsinaista kyselyä, koska varsinaiset arvot puuttuvat. Tällainen toimintatapa parantaa tietoturvan lisäksi myös suorituskykyä.

Parametrisoitujen kyselyiden avulla erotetaan kysely ja siihen liittyvä data. Kun kysely on valmiiksi käännettynä, varsinaisia arvoja ei enää käännetä SQL:läksi. Tästä syystä on mahdotonta, että hyökkääjä voisi suorittaa omia SQL-käskyään syötteensä avulla. Jos hyökkääjä esimerkiksi asettaa käyttäjänimekseen "OR 1=1", tietokannasta haetaan käyttäjää jonka käyttäjänimi on "OR1=1".

Parametrisoidut kyselyt ovat tuettuina lähes kaikissa yleisimmissä ohjelmointikielissä.

4.2 Korvaaminen ja validointi

Korvaaminen *engl. escaping* on toimenpide jossa käyttäjän syötteestä parsitaan vaaralliset merkit pois. Esimerkiksi ' merkit voidaan muuttaa \merkeiksi. Syötteen korvaamisissa on kuitenkin tietokantakohtaisia eroja. Siksi kullekin tietokannalle on olemassa omat korvaamisfunktiot. Esimerkiksi PHP:ssa käytetään MySQL:lää varten "my_sql_real_escape()" funktiota.

4.3 Datan validointi

Datan validointi ei takaa suoja SQL-injektioilta, mutta se tekee hyökkäyksestä vaikeampaa. Esimerkiksi syötteet jotka sisältää puolilainausmerkit voidaan

estää kokonaan. Toisaalta tämä ei ole aina toivottua toimintaa, esim. nimi O'Brian ei tällöin toimisi [2]. Käytössä voi olla myös luotettujen lista (*engl. whitelist*), jonne on listattu kaikki syötteelle sallitut arvot. Kaikkien sallittujen arvojen listaaminen voi kuitenkin olla vaikeaa.

4.4 Arkaluontoisen datan säilytys

4.5 Hienojakoinen pääsynhallinta tietokantaan

Tämä kappale keskittyy Roichmanin ja Gudesin artikkeliin "Fine-grained Access Control to Web Databases"[5]. Artikkelin mukaan ennen web-sovellusten yleistymistä sovelluksia ajettiin käyttäjän omalla tietokoneella. Tyypillisellä sovelluksella oli kiinteä määrä käyttäjiä. Tällaisessa sovelluksessa sovelluseros kommunikoi suoraan tietokannan kanssa. Tämän seurauksena tietokanta tietää mikä käyttäjä sitä milloinkin käyttää. Täten on helppoa rajata käyttäjien oikeuksia.

Sen sijaan web-sovelluksissa on tyypillisesti kolme kerrosta. Käyttöliittymänä toimii käyttäjän selain, joka kommunikoi web-sovelluksen palvelimen kanssa. Palvelin välittää käyttäjän käskyt tietokannalle. Tietokannan näkökulmasta komennot antaa web-sovellus, eikä komennon käyttöliittymästä lähettänyt käyttäjä. Täten tietokanta suorittaa sokeasti kaikki saamansa komennot, ellei web-sovellukseen tietokantakäyttäjän oikeuksia ole erikseen rajattu.

Aiemmin esitellyissä menetelmissä on keskitytty ratkaisemaan ongelmaa sovelluserroksella. Roichmanin ja Gudesin lähestymistavassa keskitytään ratkaisemaan ongelmaa tietokantatasolla parametrisointi metodin (*engl. parameter method*) avulla. Tekniikka perustuu parametrisoituihin näkymiin (*engl. parametrized views*). Parametrisoidun näkymän avulla voi suodattaa tallenteita ilman, että tarvitsee tehdä uutta näkymään jokaista eri parametria varten.

4.5.1 Istuntoon perustuva parametrisointi metodi

Sovelluksen tulee ylläpitää tietokantataulua johon merkataan aktiivisten käyttäjien ID:t. Roichmanin ja Gudesin esittelemä metodi toimii seuraavalla tavalla:

1. Käyttäjä kirjautuu sovellukseen ja sovellus palauttaa käyttäjälle satunnaisen AS_KEY:n, mikäli kirjautuminen onnistuu.
2. Sovellus tallettaa aktiivisten käyttäjien tauluun käyttäjän ID:n ja sitä vastaavan AS_KEY:n. Tästä lähtien kaikissa käyttäjän tekemissä SQL-kyselyissä käytetään käyttäjäkohtaista AS_KEY:tä.
3. AS_KEY poistetaan kun käyttäjä kirjautuu ulos.

Kirjautumisen jälkeen käyttäjän tiedot ovat taulussa esimerkiksi seuraavalla tavalla:

KäyttäjäID	AS_KEY
20	01010101..

Nyt voidaan käyttää seuraavanlaista parametrisoitua näkymää:

```
CREATE VIEW Palkka_View WITH pAS_KEY
SELECT * FROM Palkka
WHERE Kayttaja_ID IN
(SELECT Kayttaja_ID
FROM Kayttajat_Table
WHERE Kayttajat_Table.AS_key=:pAS_KEY)
```

Näkymä ottaa parametrina AS_Key:n, jonka käyttäjä on saanut kirjautuessaan. Käyttäjän kyselyt tehdään näkymään "Palkka_View" eikä tauluun "Palkka". Mikäli hyökkääjä yrittäisi tehdä SQL-injektion tautologian avulla, suoritettava kysely näyttäisi seuraavalta:

```
SELECT Palkka
FROM Palkka_View(01010101..)
WHERE Palkka_pvm = '12/2015' OR 1=1
```

Hyökkääjä saisi vastauksena kaikki omat palkkatietonsa, mutta ei muiden käyttäjien, koska Palkka_View saa saa parametriksi hyökkääjän oman AS_KEY:n. Myöskään UNION injektio ei ole tässä tapauksessa mahdollinen, koska hyökkääjä ei tiedä muiden käyttäjien AS_KEY:tä, joka tarvitaan Palkka_Viewiin parametriksi.

4.6 AMNESIA

Halfonding ja Orson artikkelissa [4] esitellään SQL-injektoiden torjumiseksi AMNESIA tekniikkaa. AMNESIA on lyhenne sanoille "Analysis and Monitoring for NEutralizing SQL-Injection Attacks". Tekniikka koostuu neljästä osasta.

1. Etsi suorituspaikat

Ensin ohjelman koodi skannataan. Skannauksessa etsitään koodista ne paikat, joissa tietokantakyselyjä suoritetaan. Näihin paikkoihin viitataan tässä tutkielmassa sanalla "suorituspaikka" (*engl. hotspot*). Esimerkiksi Javan tapauksessa etsitään koodista paikat joissa kutsutaan "java.sql.Statement.execute(String)" metodia.

2. Rakenna SQL-kyselymallit

Seuraavaksi rakennetaan jokaiselle edellisessä kohdassa löydetylle suorituspaikealle oma mallinsa. Tämä onnistuu siten, että AMNESIA simuloi sovelluksen toimintaa Java String Analysis (JSA) kirjaston avulla. JSA Luo analyysin tuloksena epätermistisen äärellisen automaatin (NFA), joka tunnistaa kaikki mahdolliset merkkijonot, jotka kysely voi saada arvokseen. Esimerkiksi allaoleva koodipätkä voi saada arvokseen joko: "SELECT info FROM kayttajat WHERE kayttajanimi = β " tai "SELECT info FROM kayttajat WHERE kayttajanimi='vieras'". Käyttäjän syötettä merkataan symbolilla β .

```
query = "SELECT info FROM users WHERE"
if (!kayttajanimi.empty) {
  query += " kayttajanimi =" + kayttajanimi +
    " , "
} else {
  query += " kayttajanimi = vieras "
}
```

3. Instrument application

Seuraavaksi lisätään jokaiseen vaiheessa 1. löydettyyn suorituspaikeaan monitori. Monitori suoritetaan aina ennen itse tietokantakyselyä. Monitori ottaa parametriksi suorituspaikean uniikin ID:n ja merkkijonon jota ollaan suorittamassa. ID:n avulla monitori etsii kyseistä suorituspaikeaa vastaavan mallin. Alla sama koodi esimerkkinä:

```
if (monitor.hyvaksyy(<suorituspaikean id>,
  kysely)) {
  return db.suorita(kysely);
}
```

4. Ajonaikainen monitorointi

Ajonaikana ohjelma toimii normalisti kunnes se törmää suorituspaikeaan. Suorituspaikeaan törmättyään se antaa tarvittavat parametrit monitorille. Ensin monitori käsittelee kyselyn samalla tapaa kuin tietokanta sen käsittelee. Tämän asiasta esimerkiksi erikoismerkit evaluoituvat niiden oikeaan arvoonsa. Tämä estää SQL-avainsanojen piilottamisen erikoismerkeillä. Kun kysely on käsitelty, tarkastetaan tunnistaako malli sen. Mikäli malli hyväksyy kyselyn se suoritetaan, muulloin malli tunnistaa sen SQL-injektioksi.

Oletetaan että kyselymme olisi "SELECT info FROM users WHERE kayttajanimi=" OR 1=1. Vaiheessa 1. kuvatun koodipätkän automaatti jakautuisi kahtia. Koska automaatti tunnistaa vain sellaiset kielet, jotka loppuvat merkkiin ""heti käyttäjän syötteen jälkeen, kyseinen kysely huomataan SQL-injektioksi.

TÄHÄN KUVA YLLÄOLEVASTA AUTOMAATISTA

5 Ohjelmiston testaus SQL-injektioiden varalta

Vaikka sovellusta ohjelmoitaisiin edellisen kappaleen neuvojen mukaisesti, siihen voi silti jäädä tietoturva-aukkoja. Tämän takia sovellusta on jatkuvasti tietoturvatestattava. Penetraatiotestauksessa yritetään etsiä sovelluksesta tietoturva-aukkoja. Kun penetraatiotestaus on automatisoitua, ohjelmoin ei tarvitse testata järjestelmäänsä käsin jokaisen muutoksen jälkeen. Automatisoitu testauskaan ei ole virheetöntä, sillä se voi aiheuttaa turhia hälyytyksiä, tai olla hälyyttämättä kun pitäisi hälyyttää [1]. Penetraatiotestaus voidaan jakaa kahteen alaosaan, musta laatikko -testaukseen ja valkolaatikko testaukseen.

5.1 Musta laatikko -testaus

Musta laatikko -testauksessa testataan sovellusta erilaisia syötteitä vastaan. Tällaiseen testaamiseen ei vaadita pääsyä itse koodiin [9]. Tämä on hyödyllistä esimerkiksi sellaisissa tapauksissa, kun osa ohjelman komponenteista on kolmannen osapuolen koodia, eikä siihen päästä käsiksi. Mustalaatikko testauksessa on ongelmana se, että tulos perustuu sovelluksen tulosteesta tehtyyn analyysiin. Tästä syystä kaikkia tietoturva-aukkoja ei välttämättä löydetä, tai jotain turvallista kohtaa koodista voidaan luulla tietoturva-aukoksi.

Haixia edottaa artikkelissaan "A database security testing scheme of web application"[3] seuraavanlaista testausmallia.

Ensiksi etsitään kaikki mahdolliset paikat sovelluksesta, joista käyttäjä voi syöttää dataa. Tämä onnistuu leveyssuuntaista hakua (*engl. Breadth-first search*) käyttämällä. Algoritmi toimii seuraavasti:

1. Alustetaan lista jossa on ainoana jäsenenä etusivun URL. Etusivu merkataan käsittelemättömäksi.
2. Käydään listalta läpi kaikki käsittelemättömiksi merkatut sivut. Kunkin sivun kohdalla tehdään seuraavat vaiheet:
 1. Otetaan talteen kaikki paikat joista käyttäjä voi syöttää dataa.
 2. Etsitään sivulta kaikki linkit ja lisätään listalle ne jotka eivät vielä ole siellä.
 3. Merkataan sivu käsiteltyksi.

3. Mikäli listalla on käsittelemättömiä linkkejä, palataan vaiheeseen 2.

Tämän jälkeen luodaan mahdollisimman kattava lista erilaisista haitallisista syötteistä. Kaikkiin mahdollisiin paikkoihin joista voi syöttää dataa kokeillaan kaikkia haitallisia syötteitä. Seuraavassa kappaleessa palataan siihen, kuinka haitalliset syötteet voidaan luoda. Tietokannan palauttamasta arvosta voidaan päätellä onko injektio onnistunut vai ei. Esimerkiksi jos vastauksen HTTP statuskoodi on 200, kyseessä on haavoittuvuus.

Mustalaatikkotestaukseen löytyy useita valmiita työkaluja, kuten "Acunetix Web Vulnerability Scanner" ja sqlmap.

5.1.1 Syötteiden luominen

5.2 Valkolaatikkotestaus

- static code analyzers

6 Yhteenveto

- yksinkertaista suojautaa - kuitenkin muitakin riskejä kuten xss yms. -

Lähteet

- [1] Antunes, N. ja Vieira, M.: *Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services*. Teoksessa *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim International Symposium on*, sivut 301–306, Nov 2009.
- [2] Appelt, Dennis, Nguyen, Cu Duy, Briand, Lionel C. ja Alshahwan, Nadia: *Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach*. Teoksessa *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, sivut 259–269, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2645-2. <http://doi.acm.org/10.1145/2610384.2610403>.
- [3] Haixia, Yang ja Zhihong, Nan: *A database security testing scheme of web application*. Teoksessa *Computer Science Education, 2009. ICCSE '09. 4th International Conference on*, sivut 953–955, July 2009.
- [4] Halfond, William G.J. ja Orso, Alessandro: *AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks*. Teoksessa *Georgia Institute Of Technology*.
- [5] Roichman, Alex ja Gudes, Ehud: *Fine-grained Access Control to Web Databases*. Teoksessa *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies, SACMAT '07*, sivut 31–40, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-745-2. <http://doi.acm.org/10.1145/1266840.1266846>.
- [6] Sadeghian, A., Zamani, M. ja Ibrahim, S.: *SQL Injection Is Still Alive: A Study on SQL Injection Signature Evasion Techniques*. Sept 2013.
- [7] Saldago, Roberto: *SQL Injection Optimization and Obfuscation Techniques*. 2013.
- [8] Tajpour, A., Massrum, M. ja Heydari, M.Z.: *Comparison of SQL injection detection and prevention techniques*. Teoksessa *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, nide 5, sivut V5–174–V5–179, June 2010.
- [9] Thomé, Julian, Gorla, Alessandra ja Zeller, Andreas: *Search-based Security Testing of Web Applications*. Teoksessa *Proceedings of the 7th*

International Workshop on Search-Based Software Testing, SBST 2014, sivut 5–14, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2852-4. <http://doi.acm.org/10.1145/2593833.2593835>.

- [10] Vieira, M., Antunes, N. ja Madeira, H.: *Using web security scanners to detect vulnerabilities in web services*. Teoksessa *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, sivut 566–571, June 2009.