SQL-Injektio ja siltä suojautuminen

Lalli Nuorteva

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 22. helmikuuta 2015

Sisältö

1 Johdanto

Tämän kandidaatintutkielman päätavoite on esitellä selvittää kuinka SQL-injektiolta voidaan suojautua mahdollisimman tehokkaasti. 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. Kun SQL-Injektion käsite ja lähestymistavat on esitelty, esitellään käytänteet joilla SQL-injektioita pyritään estämään.

Aihe on tärkeä koska 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. Lienee siis itsestäänselvää, että jokaisen tietokantasovelluksia ohjelmoivan on ymmärettävä mikä on SQL-injektio ja kuinka suojautua siltä.

SQL-Injektio on yleisin tietoturva-aukko web-sovelluksissa [7]. Yleisyytensä lisäksi SQL-injektio on myös hyvin vaarallinen. Onnistuneen SQL-injektion avulla hyökkääjä voi tehdä tietokannalle mitä tahansa operaatioita. SQL-Injektiolla on vuosien aikana tehty lukuisia murtoja. Yksi suurimmista murroista tehtiin SQL-Injektiolla 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 tunkeutumistestaamista varten *(engl. penetration testing)*

2 Käsitteitä

-SQL - SQL Injection out of band inband

3 SQL-Injektio

Tavallisesti relaatiotietokantoja käyttävissä sovelluksissa tietokanta on oma erillinen palvelimensa. Sovelluksella on tietokantaan omat tunnuksensa. Sovelluksen tietokantatunnuksien oikeuksia voidaan rajata halutulla tavalla. Esimerkiksi sovelluksen tietokantatunnuksilla on harvoin tarpeellista pystyä poistamaan tietokantatauluja. Sovellus kommunikoi tietokantapalvelimen kanssa käyttäen SQL (*Structured Query Language*) kyselykieltä.

SQL-Injektiossa hyökkääjä pääsee suorittamaan sovelluksen tietokannassa itse kirjoittamiaan käskyjä. Mikäli sovelluksen tietokantakäyttäjän oikeuksia ei ole rajattu, hyökkääjä voi onnistuneen SQL-injektion seurauksena suorittaa mitä tahansa tietokantapalvelimen tukemia SQL-kyselyitä. Tällöin hyökkääjän on mahdollista esimerkiksi lukea, muuttaa, lisätä tai poistaa mitä tahansa tietokannan tietoja. Jotkut tietokantapalvelimet myös sallivat käyttöjärjestelmätason komentojen 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

Anleyn artikkelin "Advancen SQL Injections in SQL Server Applications"mukaan SQL-Injektio hyökkäys esiintyy sillon, kun hyökkääjä pääsee muuttamaan käskyn logiikkaa, semantiikkaa tai syntaksia. Tämä tapahtuu lisäämällä alkuperäiseen kyselyyn uusia SQL-avainsanoja tai operaattoreita [1]. Esimerkiksi tuotteiden etsimiseen liittyvä sql-käsky voidaan rakentaa sovelluksessa seuraavalla tavalla:

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

Mikäli käyttäjä antaa nimekseen ”; DROP TABLE tuottet;’. Valmis kysely tietokannalle näyttää seuraavalta:

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

Kyseinen kysely ensin etsii 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 käsky. Esimerkiksi kaikkien tuotteiden listaamiseen hyökkääjä olisi voinut käyttää tuotenimeä joka sisältää jonkin tautologian esimerkiksi "; OR 1=!". Liittääkseen vastaukseen jonkin muun taulun hyökkääjä olisi voinut käyttää UNION:ia.

3.2 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 kyseesä 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 suorituttamalla 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-Injektiolta suojautumiseen on kehitetty useita keinoja. *TÄHÄN TIIVISTELMÄÄ ALLAOLEVISTA YMS*

4.1 Parametrisoidut kyselyt

Parametrisoiduissa kyselyssä luodaan SQL-kyselystä pohja, johon lisätään paikanpitäjät *(engl. placeholder)*. Paikanpitäjät korvataan myöhemmin varsinaisilla arvoilla. 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

Korvaaminen *engl. escaping* on toimenpide jossa käyttäjän syötteestä parsitaan vaaralliset merkit pois. Esimerkiksi ’ merkit voidaan muuttaa \merkeiksi. Syötteen korvaamisissa on kuitenkin tietokantakohtaisia eroja. Siksi kullekkin tietokannalle on olemassa omat korvaamisfunktionsa. Esimerkiksi PHP:ssa käytetään mySQL:lää varten "my_sql_real_ecape()"funktiota.

4.3 Datan validointi

Datan validointi ei takaa suojaa SQL-injektiolta, mutta se tekee hyökkäyksestä vaikeampaa. Esimerkiksi jos kyseessä on puhelinnumerokenttä, voidaan tarkistaa että syötteessä on vain numeroita. Käytössä voi olla myös luotettujen lista *(engl. whitelist)*, jonne on listattu kaikki syötteelle sallitut arvot.

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]. Ennen web-sovellusten yleistymistä sovelluksia ajettiin käyttäjän omalla tietokoneella. Tyypillisellä sovelluksella oli kiinteä määrä käyttäjiä. Tällaisessa sovelluskessa sovelluskerros kommunikoi suoraa 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 sovelluskerroksella. 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. 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 lähin 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 Menetelmä

Halfonding ja Orson artikkelissa [4] esitellään SQL-injektioiden 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 suorituspaikalle oma mallinsa. Tämä onnistuu siten, että AMNESIA simuloi sovelluksen toimintaa Java String Analysis (JSA) kirjaston avulla. JSA Luo analyysin tuloksena epädetermistisen ääreellisen automaatin (NDFA), 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 suorituspaikkaan monitori. Monitori suoritetaan aina ennen itse tietokantakyselyä. Monitori ottaa parametriksi suorituspaikan uniikin ID:n ja merkkijonon jota ollaan suorittamassa. ID:n avulla monitori etsii kyseistä suorituspaikkaa vastaavan mallin. Alla sama koodi esimerkkinä:

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

4. Ajonaikainen monitorointi

   Ajonaikana ohjelma toimii normalisti kunnes se törmää suorituspaikkaan. Suorituspaikkaan törmättyään se antaa tarvittavat parametrit monitorille. Ensin monitori käsittelee kyselyn samalla tapaa kuin tietokanta sen käsittelisi. Tämän asiosta 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 kielet jotka loppuvat merkkiin "", kyseinen kysely huomataan SQL-injektioksi.

   [tähän äskösen automaattikuva]

5 Ohjelmiston testaus SQL-injektioiden varalta

Vaikka sovellusta ohjelmoitaisiin edellisen kappaleen neuvojen mukaisesti, siihen voi silti jäädä tietoturva-aukkoja. Tämän takia sovellusta on tietoturvatestattava. Penetraatiotestauksessa yritetään etsiä sovelluksesta tietoturva-aukkoja. Kun penetraatiotestaus on automatisoitua, ohjelmoijan ei tarvitse käsin testata järjestelmäänsä jokaisen muutoksen jälkeen. Automatisoitu testauskaan ei ole virheetöntä, sillä se voi aiheuttaa turhia hälyytyksiä, tai olla hälyyttämättä kun pitäisi hälyyttää [2].

Penetraatiotestaus voidaan jakaa valkolaatikko- *(engl. white-box testing)* ja musta laatikko -testaukseen *(engl. black-box testing)*. Musta laatikko -testauksessa ei päästä käsiksi ohjelman koodiin. Ohjelmalle annetaan erilaisia syötteitä ja tutkitaan tulostetta. Tulosteesta päätellään tässä tapauksessa onko SQL-injektio tapahtunut vai ei. Valkolaatikko testauksessa sen sijaan testataan ohjelman sisäisiä struktuureja.

5.0.1 Musta laatikko -testaus

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 L jossa on ainoana jäsenenä etusivun URL. Etusivu merkataan käsittelemättökäks.

2. Käydään listalta L läpi kaikki käsittelemättömiksi merkatut sivut ja merkataan ne käsitellyiksi. Jokaiselta sivulta kirjataan kaikki paikat joista käyttäjä voi syöttää dataa. Lopuksi etsitään sivulta kaikki linkit. Ne linkit jotka eivät vielä ole listalla L, lisätään listalle.

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 sisään kokeillaan kaikkia haitallisia syötteitä. 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.

5.0.2 Valkolaatikkotestaus

- static code analyzers

6 Yhteenveto

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

Lähteet

[1] Anley, C.: *Advanced SQL Injection In SQL Server Applications.* Teoksessa *Next Generation Security Software Ltd. White Paper, 2002.*

[2] 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.

[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] Sadeghian, A., Zamani, M. ja Manaf, A.A.: *A Taxonomy of SQL Injection Detection and Prevention Techniques.* Sept 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.