SQL-Injektio ja siltä suojautuminen

Lalli Nuorteva

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 5. huhtikuuta 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Lalli Nuorteva

Työn nimi — Arbetets titel — Title

SQL-Injektio ja siltä suojautuminen

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	5. huhtikuuta 2015	16

Tiivistelmä — Referat — Abstract

Tämän kandidaatintutkielman päätavoite on selvittää kuinka SQL-injektiolta voidaan suojautua mahdollisimman tehokkaasti. Tutkielma koostuu seuraavista osa-alueista: SQL-injektion esittely, tietoturvallisen ohjelman toteutus, ajonaikainen SQL-injektioiden estäminen ja penetraatiotestaus.

Tutkielma ei rajoitu pelkästään erilaisten suojautumismetodien luetteluun, vaan osaan metodeista perehdytään myös tarkemmin. Tämä auttaa lukijaa ymmärtämään, miten valmiit ratkaisumallit toimivat. Kun ohjelmoija ymmärtää miten hänen valmitsemansa suojautumismetodit toimivat, hän tuntee myös suojauksen heikkoudet ja vahvuudet.

Avainsanat — Nyckelord — Keywords
SQL-injektio

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Viimeaikoina entistä useampi palvelu on siirtynyt verkkoon. Tämän seurauksena verkossa käsitellään jatkuvasti entistä enemmän arkaluonteisia 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 niiltä olla suojauduttu oikeaoppisesti. Tästä syystä jokaisen tietokantasovelluksia ohjelmoivan on ymmärettävä mikä on SQL-injektio ja kuinka suojautua siltä voidaan suojautua.

Tutkimusten mukaan kahdeksan prosenttia web-palveluista sisältää haavoittuvaisuuksia. 87 Prosenttia näistä haavoittuvuuksista on SQL-injektio haavoittuvuuksia [VAM09]. 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-Injektio mielletään monesti amatöörien ongelmaksi, mutta sen avulla on murrettu myös paljon ammattilaisten toteuttamia järjestelmiä. Yksi suurimmista SQL-Injektion avulla tehdyistä murroista tehtiin Guess.com:ille [Pou02]. Hyökkääjä sai käsinsä 200 000 ihmisen nimet ja luottokorttitiedot.

Sen lisäksi että SQL-injektio on yleisin tietoturva-aukko, se on myös helppoa toteuttaa ilman syvällistä ymmärrystä sen toiminnasta. Esimerkiksi penetraatiotestaustyökalut , kuten "sqlmap" , ilmoittavat sivuston heikkouksista melkeinpä napin painalluksella. Vaikka sqlmapin kaltaiset työkalut onkin tehty nimenomaan penetraatiotestaukseen, mikään ei estä hyökkääjää käyttämästä niitä apuna hyökkäyksissä. Niinpä on tärkeää, että ohjelmoija tuntee yleisimmät penetraatiotestaustyökalut, sekä käyttää niitä.

Toisaalta SQL-injektota ei ole vaikeaa estää, mikäli ohjelmoija ymmärtää kuinka SQL-injektio toimii. Nykyaikaiset web-applikaatioiden viitekehykset, sekä ohjelmointikielet tarjoavat työkaluja SQL-injektioiden torjumiseen. Työkalujen käytöstä on tehty niin yksinkertaista, että ohjelmoijan tarvtisee vain muistaa käyttää niitä oikeissa paikoissa lähdekoodia. Esimerkiksi Ruby on Rails:issa "Model.find_by_something(parametri)" hoitaa itsestään parametrin tarkastamisen ja käsittelyn SQL-injektoin varalta. SQL-Injektion estämisestä on tehty jopa niin helppoa, että jotkut saattavat erehtyä luulemaan, ettei ohjelmoijan tarvitse enää huolehtia siitä. Kuitenkin esimerkiksi Railsin "Model.where(parametri)" -metodi on altis SQL-injektoille. Niinpä ohjelmoija ei voi luottaa viitekehyksen tai ohjelmointikielen hoitavan automaattisesti kaikkea. Tästä syystä ymmärrys SQL-injektioiden toiminnasta on yhä nykyaikanakin välttämätöntä.

2 SQL-Injektio

Anleyn artikkelin "Advancen SQL Injections in SQL Server Applications"[Anl02] mukaan SQL-Injektio hyökkäys esiintyy sillon, 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 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.

2.1 SQL-Injektio käytännössä

Esimerkiksi tuotteiden etsimiseen liittyvä SQL-käsky voidaan rakentaa SQL-injektiolle alttiissa 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 etsii ensin kaikki tuotteet joiden nimi on tyhjä. Seuraavaksi suoritetaan komento "DROP TABLE tuotteet", joka poistaa koko tuotteet taulun.

Edellä mainitun hyökkäyksen sijaan hyökkääjä olisi voinut käyttää esimerkiksi jotakin Srivastavan artikkelissa "Algorithm to prevent back end database against SQL injection attacks" [Sri14] esitellyistä menetelmistä:

Tautologia

> Hyökkääjä voi käyttää tuotenimeä, joka sisältää jonkin tautologian esimerkiksi "; OR 1=1". Tällöin hyökkääjä olisi saanut vastauksena kaikki tuotteet.

Kommentti

Mikäli tuote olisi vaatinut myös tuotekoodin, eli kyselyn muotoilu olisi ollut esimerkiksi seuraavanlainen:

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

Hyökkääjä voi kirjoittaa tuotenimeksi "Haluttu tuote'; –". Hyökkäyksen onnistuessa loput kyselystä muuttuu kommentoiduksi. Tällöin tuotteen tiedot palautetaan, vaikka hyökkääjä ei tietäisi tuotekoodia.

Union kysely

Hyökkääjä voi kirjoittaa tuotenimeksi esimerkiksi "Haluttu tuote' UNION SELECT * FROM kayttajatiedot". Hyökkäyksen onnistuessa vastaukseen sisältyy myös koko taulun "kayttajatiedot" sisältö.

2.2 Obfuskointi

Edellisen kappaleen esimerkit eivät todennäköisesti toimi sovelluksissa, jotka on suojattu SQL-injektioilta. Tästä syystä hyökkääjien on keksittävä tapoja, joilla palomuureja voidaan yrittää kiertää. Salgadon kirjoittaman "SQL Injection Optimization and Obfuscation Techniques"[Sal13] artikkelissa esitellään obfuskointia. Obfuskointi tarkoittaa koodin tahallista monimutkaistamista ja epäselkeyttämistä sen varsinaisen toiminnan piilottamiseksi. Obfuskointia käytetään esimerkiksi haittaohjelmien piilottamiseen virustutkilta. SQL-Injektoiden tapauksessa obfuskointi voi olla yksinkertaisimillaan esimerkiksi "DROP" avainsanan muuttaminen "DroP":iksi. Tällöin yksinkertainen mustalistaukseen perustuva palomuuri saattaisi päästää SQL-injektion läpi.

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 palomuuri 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".

2.3 SQL-Injektioiden luokittelu

Sadeghiani ja hänen kollegoidensa artikkelin "SQL-Injection is still alive" mukaan SQL-injektioita luokitallaan sen perusteella, mitä reittiä hyökkääjä saa

palautteen sovellukselta. SQL-injektioita on kolmea eri päätyyppiä, inband injektio, out-of-band injektio ja sokea injektio [SZI13].

2.3.1 Inband injektio

Artikkelin mukaan inband tyyppinen injektio on kyseessä silloin, kun vastaus saadaan samaa reittiä, jota hyökkäys on suoritettu. Tällainen hyökkäys onnistuu silloin, kun kyselyn tulos palautetaan suoraan käyttäjälle. Tällainen aukko saattaa esiintyä esimerkiksi sovelluksessa, jossa hyökkääjä voi hakea ystäviensä lisäämiä kuvia. Hyökkäyksen tuloksena hyökkääjä voisi saada vaikkapa kaikki tietokannasta löytyvät kuvat.

2.3.2 Out-of-band injektio

Sen sijaan out-of-band injektiossa tuloste saadaan eri reittiä, kun hyökkäys on syötetty. Out-of-band injektiota voidaan hyödyntää, vaikka sovellus ei palauta käyttäjälle kyselyn tulosta. Tällainen tilanne voi esimerkiksi ilmetä, jos sovellus tallettaa tietokantaan käyttäjien selaintietoja. Selaintiedot saadaan HTTP-pyynnön "User-Agent" kentästä. Hyökkääjä voisi tällaisessa tapauksessa lähettää "User-Agent" kentässä haitallista koodia. Jotta hyökkääjä saisi vastauksen suoritetusta kyselystä, hän voi ohjata vastauksen esimerkiksi oman palvelimensa logeihin. Tämä onnistuu, mikäli sovellus saadaan suorittamaan esimerkiksi seuraavanlainen kysely:

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

Oletetaan että käyttäjän admin salasana on "password123". Injektion onnistuessa tietokanta hakee ensin käyttäjän admin salasanan, jonka jälkeen se suorittaa HTTP-pyynnön osoitteeseen "http://hyokkaajansivu.fi/injections/password123". Tällöin hyökkääjän palvelimen logeissa näkyy seuraavanlainen merkintä:

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

Sadeghianin ja kollegoiden artikkelin mukaan kyselyn tulos voidaan ohjata palvelimen lisäksi myös esimerkiksi hyökkääjän sähköpostiin.

2.3.3 Sokea injektio

Artikkelin mukaan sokeassa injektiossa hyökkääjä ei saa kyselyn palauttamaa tulosta selville mitään reittiä. Hyökkääjä voi päätellä hyökkäyksen onnistumisen esimerkiksi siitä 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 [TMH10]. Hyökkääjä voi myös tarkkailla vaikuttaako hänen tekemänsä hyökkäykset sovelluksen toimintaan tai ulkoasuun jollain tapaa.

3 Tietoturvallisen ohjelman toteutus

3.1 Käyttäjän syötteen käsittely

SQL-Injektiolta välttymiseksi käyttäjän syötteet on tarkastettava. Vaaralliseksi epäilty syöte voidaan käsitellä vaarattomaksi, tai jättää suorittamatta kokonaan. Tässä kappaleessa esitellään kolme erilaista tapaa käsitellä käyttäjän syötettä: datan validointi, korvaaminen ja parametrisoidut kyselyt.

3.1.1 Validointi

Validointi voidaan toteuttaa joko musta- tai valkolistan avulla tai vertaamalla syötettä erilaisiin säännöllisiin lauseisiin. Valkolistan tapauksessa kaikki sallitut syötteet on kirjattuna valmiiksi. Tällainen lähestymistapa on kuitenkin usein mahdoton, koska valkolista kasvaisi liian suureksi. Mustalistalla voidaan pyrkiä listaamaan kaikki kielletyt syötteet tai merkit, mutta mustalista saattaa on kierrettävissä obfuskoinnin avulla [Sal13].

Käyttäjän syötettä voidaan myös verrata säännölliseen lausekkeeseen, esimerkiksi puhelinnumeokenttä voidaan rajoittaa hyväksymään pelkästään numeroita. Syöte voidaan myös tarkastaa SQL-avainsanojen tai erikoismerkkien varalta. Tällainen lähestymistapa voi kuitenkin rajoittaa myös päteviä syötteitä. Esimerkiksi käyttäjänimi "O'Brian"olisi estetty ""merkin vuoksi [ANBA14a].

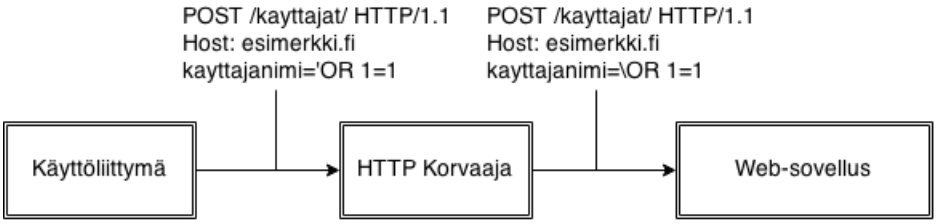
3.1.2 Korvaaminen

Datan validoinnin sijaan voidaan käyttää korvaamista (*engl. escaping*). Korvaamisessa vaarallisia syötteitä ei hylätä. Hylkäämisen sijaan kaikille syötteille suoritetaan korvausoperaatio. Korvausoperaatiossa haitalliset erikoismerkit, kuten ' korvataan joillakin vaarattomilla merkeillä kuten V̇aarallisissa merkeissä on kuitenkin tietokantakohtaisia eroja. Tästä syystä ohjelmoijan tulee käyttää tietokantakohtaista korvausfunktiota. Tyypillisesti korvausfunktiota ei ole tarpeellista toteuttaa itse, sillä ohjelmointikielet tarjoavat niihin usein valmiita toteutuksia. Esimerkiksi PHP:ssa on mySQL tietokantaa varten luotu "my_sql_real_ecape_string()" funktio.

Sadeghanin ja Zamanin artikkelin "SQL injection vulnerability general patch using header sanitization" [SZAM14] mukaan korvaamista voidaan käyttää myös HTTP-pakettien käsittelyyn. Web-sovellukset saavat tyypillisesti viestinsä HTTP-protokollaa käyttäen. Saapuvat HTTP-paketit voidaan käsitellä korvausfunktion avulla ennen kun ne annetaan itse sovellukselle käsiteltäväksi. Tällä tavoin vanhasta sovelluksesta voidaan yrittää tehdä tietoturvallinen ilman, että sen lähdekoodia tarvitsee muokata. Artikkelin mukaan esimerkiksi PHP:n tapauksessa metodin käyttöönotto vaatii vain yhden kirjaston käyttöönottoa. Tämän ansiosta metodin käyttöönotto on todennäköisesti nopeampaa, kuin koko sovelluksen refaktorointi SQL-injektioiden varalta. Toisaalta erilisen HTTP-pakettien käsittelijän lisääminen voi huonontaa ohjelman suorituskykyä.

Korvaaminen ei kuitenkaan ole aina toivottua. Esimerkiksi nimimerkki "O'Brian" korvautuu "O\Brian:iksi", joka ei todennäköisesti ole toivottavaa.

Kuva 1: Syöte voidaan korvata erillisessä HTTP-paketti käsitelijässä jo ennen kuin paketti annetaan sovellukselle..



3.1.3 Parametrisoidut kyselyt

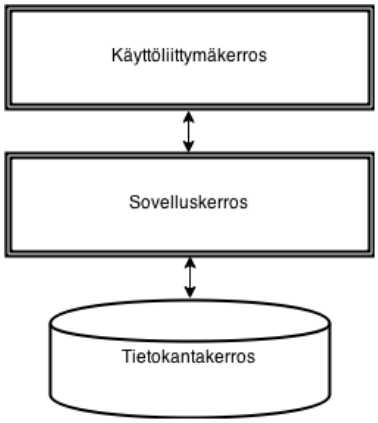
Parametrisoidut kyselyt esitellään artikkelissa "SQL injection vulnerability general patch using header sanitization" [SZAM14]. Niiden avulla voidaan tallentaa käyttäjän antama syöte sellaisenaan tietokantaan, ilman että se muuttaa valmiiksi luodun kyselyn syntaksia. Tällöin "O'Brian"tallentuu "O'Brian":ina juuri kuten pitääkin. Parametrisoidussa kyselyssä luodaan SQL-kyselyistä pohjia, joihin lisätään paikanpitäjät (*engl. placeholder*). Parametrisoitu kysely annetaan tietokannalle. Tietokanta kääntää ja optimoi kyselyn pohjan vain kerran. Tällainen toimintatapa parantaa suorituskykyä. Tietokanta ei kuitenkaan suorita varsinaista kyselyä heti, koska siitä puuttuu haettavat arvot. Kun käyttäjä antaa syötteenä haluamansa arvot, paikanpitäjät korvataan arvoilla. Käyttäjän antamia arvoja ei enää tulkata SQL:lläksi, vaan niitä käsitellään sellaisinaan. Tällöin ei ole mahdollista että käyttäjän syöte rikkoisi kyselyä. Esimerkiksi jos käyttäjä antaa käyttäjänimekseen "; OR 1=1", niin tietokannasta haetaan käyttäjää, jonka käyttäjänimi on "OR 1=1". Parametrisoidut kyselyt ovat tuettuina lähes kaikissa yleisimmissä ohjelmointikielissä [jav15]. Oikein käytettynä parametrisoidut kyselyt suojaavat sovelluksen täysin SQL-injektioilta [SZAM14].

3.2 Koodikatselmointi

Antunesin ja kumppaneiden tekemässä vertailussa[AV09] vertailtiin staattisen koodianalyysin ja penetraatiotestaamisen eroja. Kumpikaan testaustapa ei löytänyt yli 51% sovelluksen tietoturva-aukoista. Tällaisten aukkojen huomaaminen on kuitenkin mahdollista, kun ohjelman lähdekoodia katselmoi useammat henkilöt.

3.3 Hienojakoinen pääsynhallinta tietokantaan

Tämä kappale keskittyy Roichmanin ja Gudesin artikkeliin "Fine-grained Access Control to Web Databases"[RG07]. Artikkelin mukaan 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.



Nykyään web-sovelluksissa on tyypillisesti kolme kerrosta [RG07][APG+12]. 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.

Sovelluksen tulee ylläpitää tietokantataulua johon merkataan aktiivisten käyttäjien ID:t. Roichamin 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 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 Ajonaikainen SQL-injektoiden estäminen

SQL-Injektioida voidaan yrittää havaita ja estää ajonaikana. Tätä varten on luotu useita työkaluja, kuten SQLCheck, SQLProb ja Candid. [ST13]. Ajonaikaiseen SQL-injektioiden estämiseen on kehitetty monenlaisia tapoja. Tässä kappaleessa perehdytään tarkemmin AMNESIA ja SQLrand menetelmiin.

4.1 AMNESIA

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 kayttajat 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:

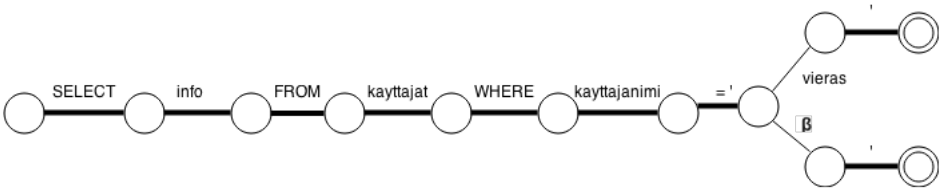
```
if (monitor.hyvaksyy(<suortuspaikan id>,
    kysely)) {
return db.suorita(kysely);
}
```

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 sellaiset kielet, jotka loppuvat merkkiin " ’ " heti käyttäjän syötteen jälkeen, kyseinen kysely huomataan SQL-injektioksi.

Kuva 2: Vaiheen 2. koodia vastaava automaatti



4.2 SQLRand

Boyd ja Keromytis esittelevät SQLRand menetelmän SQL-injektiota vastaan artikkelissaan "SQLRand: Preventing SQL Injection Attacks" [BK04]. SQLRandissa on ideana lisätä jokaiseen SQL-avainsanaan jokin satunnainen numero. Mikäli hyökkääjä yrittäisi suorittaa SQL-injektion tavallisilla SQL-avainsanoilla, se epäonnistuihi, koska sovellus ei tunnistaisi sitä SQL:läksi.

Tavallinen tietokanta ei ymmärrä satunnaistettuja SQL-avainsanoja. Artikkelissa tähän ongelmaan oli luotu kaksi eri ratkaisua. Ensimmäinen on muokata tietokannan SQL-tulkkia siten, että se ymmärtää muokattuja SQL-avainsanoja. Tämä voi kuitenkin olla vaikeaa toteuttaa, sekä se saattaa sotkea muiden sovellusten toimintaa, jotka käyttävät samaa tietokantapalvelinta. Toinen ratkaisu on luoda erillinen tulkkisovellus tietokantapalvelimen ja web-sovelluksen välille.

SQLRandia on kritisoitu sen monimutkaisuudesta [SZAM14]. Kritisoijat pitävät SQLRandin toteuttamista olemassa olevaan sovellukseen aikaavievänä. Sen katsotaan myös hankaloittavan sovelluksen käyttöönottoa.

5 Penetraatiotestaus

Vaikka sovellusta ohjelmoitaisiin hyvien ohjelmointikäytänteiden mukaisesti, siihen voi silti jäädä tietoturva-aukkoja. Tämän takia sovellusta on jatkuvasti tietoturvatestattava.

Penetraatiotestauksessa yritetään etsiä sovelluksesta tietoturva-aukkoja. Kun penetraatiotestaus on automatisoitua, ohjelmoijan ei tarvitse suorittaa samoja testausruutineja jokaisen muutoksen jälkeen. Penetraatiotestaus ei kuitenkaan ole virheetöntä. Artikkelissa "Using web security scanners to detect vulnerabilities in web services"[VAM09] kerrotaan, että penetraatiotestaus voi aiheuttaa turhia hälyytyksiä, tai olla hälyyttämättä kun pitäisi hälyyttää. Artikkelin mukaan jopa yli 30% havaituista virheistä osoittautuivat virheellisiksi. Tämän takia penetraatiotestauksen tulee olla vain yksi testaustyökaluista, eikä siihen voida sokeasti luottaa.

Musta laatikko -testauksessa testataan sovellusta erilaisia syötteitä vastaan. Tällaiseen testaamiseen ei vaadita pääsyä itse koodiin [TGZ14]. 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, eikä esimerkiksi tietokannan todelliseen tilaan. Tästä syystä kaikkia tietoturva-aukkoja ei välttämättä löydetä, tai jotain turvallista kohtaa koodista voidaan luulla tietoturva-aukoksi. Mustalaatikkotestaukseen löytyy useita valmiita työkaluja, kuten "Acunetix Web Vulnerability Scanner" ja sqlmap.

Valkolaatikko testauksessa testaajalla on pääsy ohjelman lähdekoodiin. Valkolaatikko testaustyökaluihin kuuluu esimerkiksi staattiset lähdekoodin analysointi työkalut. Tällaisilla työkaluilla voidaan huomata mahdollisia tietoturva-aukkoja jo ennen kun ohjelmaa on ajettu ensimmäistäkään kertaa [GGS14].

5.1 Penetraatiotestauksen toiminta

Haixia edottaa artikkelissaan "A database security testing scheme of web application"[HZ09] 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 käsittelemättökäks.

2. Käydään listalta läpi kaikki käsittelemättömiksi merkatut sivut. Kunkin sivun kohdalla tehdään seuraavat vaiheet:

   - Otetaan talteen kaikki paikat joista käyttäjä voi syöttää dataa.

   - Etsitään sivulta kaikki linkit ja lisätään listalle ne jotka eivät vielä ole siellä.

   - Merkataan sivu käsitellyksi.

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

Tämän jälkeen luodaan mahdollisimman kattava lista erilaisista hyökkäyksistä. Kaikkiin mahdollisiin paikkoihin joista voi syöttää dataa kokeillaan haitallisia syötteitä. Tietokannan palauttamasta arvosta voidaan päätellä onko injektio onnistunut vai ei. Esimerkiksi jos vastauksen HTTP statuskoodi on 200, kyseessä on haavoittuvuus.

5.2 Testaussyötteiden luominen

Artikkelissa "Automated Testing for SQL Injection Vulnerabilities: An input Mutation Approach"[ANBA14b] ehdotetaan penetraatiotestauksessa käytettävien syötteiden luomiseen automatisoitua tekniikkaa nimeltään μSQLi. Tekniikassa on ideana manipuloida kelpaavaa syötettä erilaisilla mutaatio-operaatioilla. Mutaatio-operaatiot jaetaan artikkelin mukaan seuraavalla tavalla kolmeen eri osioon:

1. Käyttäytymistä muuttavat operaatiot

   Esimerkiksi operaatiot jotka lisäävät AND tai OR lauseen, tai operaatiot joissa lisätään puolipiste ja kokonaan uusi SQL lause.

2. Syntax-Repairing Operators

   Lisää kyselyyn esimerkiksi sulut, kommenttimerkin tai heittomerkin.

3. Obfuskointi operaatiot

   Esimerkiksi muuttaa kyselyssä käytettävää enkoodausta tai muuttaa totuuslauseketta ilman että sen arvo muuttuu.

Toimivaan syötteeseen voidaa lisätä yksi tai useampi mutaatio-operaatio. Artikkelin mukaan useasti yksittäinen operaatio huomataan, mutta yhdistelmät saattavat silti jäädä huomaamatta. Mutaatioiden tekeminen aloitetaan toimivasta syötteestä, koska sillä vältetään se, että syöte hylättäisiin välittömästi. Lisäksi toimivat syötteet täyttävät todennäköisemmin syötevalidoinnit.

6 Yhteenveto

(pahasti kesken) - osalla saadaan vanhoista ohjelmista toimivia

SQL-Injektioilta suojautuminen voidaan jakaa ohjelmointikäytänteisiin, ajonaikaiseen monitorointiin ja testaukseen. Periaatteessa jo hyvät ohjelmointikäytänteet riittävät sovelluksen suojautumiseen SQL-injektiolta. Ei kuitenkaan voida olla varmoja muistetaanko hyviä ohjelmointikäytänteitä aina noudattaa. Tästä syystä on sovellusta tulee penetraatiotestata. Penetraatiotestauksessa taas on ongelmana sen luotettavuus. Heikon luotettavuuden takia ei voida koskaan olla varmoja onko sovellus täysin turvallinen. Mitä enemmän menetelmiä käyttää, sitä turvallisempi sovellus on. Menetelmien yhdistely on kuitenkin aikaa vievää. Lisäksi osa menetelmistä tekee sovelluksesta, tai sen tuottamisesta hitaampaa ja monimutkaisempaa. On siis vaikeaa sanoa milloin sovellus on tarpeeksi hyvin suojattu.

SQL-injektio ei kuitenkaan ole ainut tietoturvariski.

Lähteet

[ANBA14a] 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`.

[ANBA14b] 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`.

[Anl02] Anley, C.: *Advanced SQL Injection In SQL Server Applications.* Teoksessa *Next Generation Security Software Ltd. White Paper, 2002.*, 2002.

[APG+12] Avireddy, S., Perumal, V., Gowraj, N., Kannan, R.S., Thinakaran, P., Ganapthi, S., Gunasekaran, J.R. ja Prabhu, S.: *Random4: An Application Specific Randomized Encryption Algorithm to Prevent SQL Injection.* Teoksessa *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, sivut 1327–1333, June 2012.

[AV09] 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.

[BK04] Boyd, Stephen W. ja Keromytis, Angelos D.: *SQLrand: Preventing SQL Injection Attacks.* Teoksessa *In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, sivut 292–302, 2004.

[GGS14] Gupta, M.K., Govil, M.C. ja Singh, G.: *Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey.* Teoksessa *Recent Advances and Innovations in Engineering (ICRAIE), 2014*, sivut 1–5, May 2014.

[HZ09] 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.

[jav15] *Using Prepared Statements.* Tarkasteltu: 02.03.2015. `http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html`.

[Pou02] Poulsen, Kevin: *Guesswork Plagues Web Hole Reporting.* March 2002. `http://www.securityfocus.com/news/346`.

[RG07] 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`.

[Sal13] Saldago, Roberto: *SQL Injection Optimization and Obfuscation Techniques.* 2013.

[Sri14] Srivastava, M.: *Algorithm to prevent back end database against SQL injection attacks.* Teoksessa *Computing for Sustainable Global Development (INDIACom), 2014 International Conference on*, sivut 754–757, March 2014.

[ST13] Shar, L.K. ja Tan, Hee Beng Kuan: *Defeating SQL Injection*, nide 46. March 2013.

[SZAM14] Sadeghian, A., Zamani, M. ja Abd Manaf, A.: *SQL injection vulnerability general patch using header sanitization.* Teoksessa *Computer, Communications, and Control Technology (I4CT), 2014 International Conference on*, sivut 239–242, Sept 2014.

[SZI13] Sadeghian, A., Zamani, M. ja Ibrahim, S.: *SQL Injection Is Still Alive: A Study on SQL Injection Signature Evasion Techniques.* Sept 2013.

[TGZ14] 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`.

[TMH10] 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.

[VAM09] 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.