Mika Lehtinen

Haavoittuvuuksien torjunta defensiivisillä ohjelmointikeinoilla PHP-sovelluksissa

Tietotekniikan
kandidaatintutkielma
5. joulukuuta 2012

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Mika Lehtinen
Yhteystiedot: mika.k.lehtinen@student.jyu.fi
Työn nimi: Haavoittuvuuksien torjunta defensiivisillä ohjelmointikeinoilla PHP-sovelluksissa
Title in English: Preventing vulnerabilities in PHP applications using defensive programming techniques
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 20
Tiivistelmä: Tutkielmassa kartoitetaan, kuinka SQL-injektioita, Cross-Site Scripting -hyökkäyksiä ja Cross-Site Request Forgery -hyökkäyksiä voidaan ehkäistä PHP-sovelluksissa defensiivisillä ohjelmointimenetelmillä.
Abstract: This thesis explores the defensive programming techniques for preventing SQL injections, Cross-Site Scripting attacks and Cross-Site Request Forgery attacks in PHP applications.
Avainsanat: PHP, SQL-injektio, cross-site scripting, cross-site request forgery
Keywords: PHP, SQL injection, cross-site scripting, cross-site request forgery

Sisältö

1 Johdanto

Nykyään yhä useammat yritykset ja organisaatiot pyrkivät kehittämään liiketoimintaansa ottamalla käyttöön erityyppisiä WWW-sovelluksia. Erityisesti dynaamisten verkkosovellusten käyttö on lisääntynyt. Nämä sovellukset käyttävät yleensä tietokantaa, johon tallennetaan asiakkaiden tietoja. Kyseiset tiedot ovat usein hyvin arkaluontoisia, joten yritysten on huolehdittava, että ne pysyvät salassa. Tämä ei kuitenkaan ole yksinkertaista, sillä WWW-sovelluksiin jää helposti haavoittuvuuksia, joita hyökkääjät voivat käyttää hyväkseen. Verkkohyökkäyksen onnistuessa yritykselle voi aiheutua mittavat vahingot.

SQL-injektio on yksi vakavimmista WWW-sovellusten haavoittuvuuksista. SQL-injektiossa hyökkääjä syöttää sovellukseen sellaista dataa, jonka WWW-sovellus tulkitsee osittain SQL-lauseeksi [11]. Tämä saattaa antaa hyökkääjälle pääsyn koko WWW-sovelluksen tietokantaan, mikä mahdollistaa esimerkiksi identiteettivarkauden tai petoksen [11]. Vaikka SQL-injektio on ehkä tunnetuin WWW-sovellusten haavoittuvuustyyppi, monista WWW-sovelluksista on silti löydetty hiljattain uusia haavoittuvuuksia [16].

Toinen merkittävä WWW-sovellusten haavoittuvuus on *Cross-Site Scripting* (XSS). Esimerkiksi vuonna 2006 XSS oli WWW-sovellusten suurin haavoittuvuustyyppi julkisesti ilmoitetuista haavoittuvuuksista [6]. XSS-haavoittuvuudella tarkoitetaan sitä, että WWW-sovellus tulostaa käyttäjälle jonkun toisen käyttäjän syötettä tarkistamatta sitä. Syöte voi sisältää JavaScript-koodia, jolloin käyttäjän avatessa sivun koodi aktivoituu. XSS-hyökkäykset kohdistuvat siis siihen kohtaan WWW-sovellusta, joka tulostaa HTML-koodia selaimelle [17]. Myös XSS-hyökkäyksellä voi olla vakavia seurauksia, kuten käyttäjätilin kaappaaminen [17].

Cross-site request forgery (CSRF) tapahtuu, kun haitallinen WWW-sivusto tekee käyttäjän tietämättä haitallisen pyynnön toiselle, käyttäjän luottamalle sivustolle [20]. Vaikka CSRF-hyökkäykset ovat saaneet verraten vähän huomiota [5], ne ovat silti vakava tietoturvariski. Esimerkiksi Zeller ja Felten [20] löysivät `ingdirect.com`-sivustolta CSRF-haavoittuvuuden, joka mahdollisti rahansiirron käyttäjän tililtä hyökkääjän tilille.

Tutkielmassa kartoitetaan, miten edellä mainittuja haavoittuvuuksia voidaan ehkäistä PHP-sovelluksissa defensiivisten ohjelmointikeinojen avulla. Defensiiviset ohjelmointimenetelmät ovat esimerkiksi paras keino ehkäistä XSS-hyökkäyksiä [17] sekä SQL-injektioita [11].

Kirjallisuuskatsauksessa löytyneissä artikkeleissa defensiivisiä ohjelmointikeinoja esiteltiin yleisesti teoreettisella tasolla, mutta niiden käytännön toteutukseen ei

juurikaan otettu kantaa. Lisäksi useimmissa artikkeleissa pääpaino oli sillä, kuinka jo olemassa olevasta koodista voidaan tunnistaa haavoittuvuuksia automaattisilla menetelmillä tai kuinka hyökkäyksiä voidaan torjua ajon aikana.

Tällaisissa automaattisissa menetelmissä on kuitenkin ongelmia. Ensinnäkin niitä voi olla käytännössä vaikea toteuttaa, minkä vuoksi niihin saattaa jäädä ohjelmointivirheitä. Tämän takia ohjelmoijien ei pitäisi täysin sokeasti luottaa automaattisiin työkaluihin, sillä ne eivät välttämättä huomaa kaikkia mahdollisia haavoittuvuuksia. Lisäksi koodinanalysointityökalut voivat antaa vääriä hälytyksiä, mikä voi hidastaa ohjelmointityötä [2]. Automaattisten menetelmien käyttäminen ei todennäköisesti myöskään edistä ohjelmoijien tietoutta siitä, mistä haavoittuvuudet todella johtuvat ja miten niitä voisi helpoimmin ehkäistä ilman automaattisia työkaluja. Esimerkiksi SQL-injektiot johtuvat monesti siitä, että kehittäjät käyttävät väärin ohjelmointikielen tarjoamia keinoja SQL-kyselyiden suorittamiseen [16].

Tutkielman aiheena on selvittää, miten XSS-, CSRF- ja SQL-injektiohaavoittuvuuksia voidaan käytännössä ehkäistä PHP-sovelluksissa defensiivisten ohjelmointikeinojen avulla. Seuraavat ovat tarkentavia kysymyksiä: Millaisia ohjelmointikeinoja kunkin haavoittuvuustyypin ehkäisemiseksi on olemassa? Mitkä ovat kyseisten keinojen vahvuudet ja heikkoudet? Kuinka parhaat keinot voidaan käytännössä toteuttaa PHP-sovelluksissa?

Tutkielmassa keskitytään PHP-sovelluksiin, koska PHP on hyvin laajalle levinnyt ohjelmointikieli dynaamisten WWW-sovellusten teossa. Esimerkiksi vuoteen 2007 mennessä PHP oli käytössä yli 20 miljoonalla WWW-sivustolla [18].

2 Defensiiviset ohjelmointikeinot kirjallisuudessa

Kaiken kaikkiaan kirjallisuuskatsauksen tuloksena löytyi kuusi artikkelia. Yhteistä näille artikkeleille on se, että niissä esitellään defensiivisiä ohjelmointikeinoja XSS-, CSRF- ja SQL-injektiohaavoittuvuuksien ehkäisemiseksi. Tässä kontekstissa defensiivisillä ohjelmointimenetelmillä tarkoitetaan niitä ohjelmointikielen keinoja, joilla WWW-sovellusten haavoittuvuuksia voidaan ehkäistä ennalta.

Halfond, Viegas ja Orso [11] esittelevät artikkelissaan tunnetut SQL-injektiotyypit. Tutkimuksessa annetaan esimerkit siitä, kuinka hyökkääjät voivat hyödyntää erityyppisiä injektiohaavoittuvuuksia sekä miten SQL-injektioita vastaan voi suojautua. Näitä suojausmekanismeja arvioidaan sen mukaan, kuinka hyvin ne tehoavat eri injektiotyyppejä vastaan. Erityisesti artikkelissa esitellään neljä defensiivistä ohjelmointikeinoa SQL-injektioiden varalle. Näitä ovat esimerkiksi syötteen tyypin tarkistaminen, syötteen koodaus sekä sallittuihin merkkeihin perustuva suodatus

(*positive pattern matching, whitelist filtering*), joita myös Shar ja Tan [16] ehdottavat. Artikkelissa esitellään myös SQL-injektioiden automaattisia tunnistus- ja torjuntatekniikoita sekä arvioidaan niiden vahvuuksia ja heikkouksia. Yleisimpänä heikkoutena todetaan se, että tekniikat pystyvät tunnistamaan vain tietyntyyppisiä haavoittuvuuksia jättäen muut huomiotta. Toisena merkittävänä haittana todetaan suuri väärien hälytysten määrä.

Shar ja Tan [16] tekevät artikkelissaan katsauksen niihin tekniikkoihin, joilla SQL-injektioita voidaan torjua sekä kertovat syitä sille, miksi WWW-sovelluksiin jää SQL-injektiohaavoittuvuuksia. Näitä ovat heidän mukaansa esimerkiksi syötteen tarkistamatta jättäminen, vääränlainen erotinmerkkien käyttäminen SQL-kyselyissä ja riittämätön syötteen siivoaminen. Lisäksi artikkelissa esitellään defensiivisiä keinoja injektioiden torjumiseksi, kuten parametrisoitujen SQL-kyselyjen käyttö. Tällaisissa kyselyissä SQL-lausetta ei muodosteta liittämällä merkkijonoja yhteen, vaan SQL-lauseeseen jätetään parametreja varten paikkamerkit, ja lopullinen SQL-kysely muodostetaan yhdistämällä käyttäjän syöttämät parametrit näihin paikkoihin.

Shar ja Tan [17] esittelevät toisessa artikkelissaan XSS-hyökkäysten perusidean ja XSS-injektiotyypit, joita on kolmenlaisia: pysyviä, ei-pysyviä sekä DOM-pohjaisia. Pysyvässä XSS-injektiossa haitallista koodia tallentuu WWW-sovelluksen tietokantaan, kun taas ei-pysyvässä eli heijastuneessa XSS-injektiossa WWW-sovellus tulostaa (ts. heijastaa) käyttäjän antaman syötteen takaisin käyttäjälle tarkistamatta sitä. DOM-pohjaiset XSS-haavoittuvuudet puolestaan johtuvat WWW-sivun skriptien käsitellessä käyttäjän syötettä huolimattomasti. Artikkelissa kerrotaan myös esimerkkejä XSS-hyökkäyksistä sekä tavat, joilla XSS-haavoittuvuuksia voidaan ehkäistä. Kyseiset keinot ovat hyvin samankaltaisia kuin SQL-injektioiden tapauksessa. Käyttäjän antamasta syötteestä voidaan esimerkiksi etsiä kiellettyjä merkkejä, jotka voidaan joko vaihtaa vaarattomiksi merkeiksi (*escape*) tai poistaa kokonaan.

Antunes ja Vieira [2] kertovat yleisellä tasolla siitä, miten WWW-sovellusten haavoittuvuuksia voi ehkäistä. Heidän mukaansa WWW-sovellukset tarvitsevat kolmea erityyppistä suojausta: syötteen tarkistusta (*input validation*), tulosteen tarkistusta (*output validation*) sekä ohjelmakoodin kriittisten paikkojen suojausta (*hotspot protection*). Syötteen tarkistus tarkoittaa kaiken sellaisen datan tarkistamista, joka WWW-sovellukselle tulee ulkoapäin (yleensä käyttäjältä). Tulosteen tarkistus puolestaan tarkoittaa käyttäjän selaimelle lähetettävän tiedon tarkistamista. Tulosteen tarkistuksella on mahdollista ehkäistä XSS-haavoittuvuuksia sekä tietovuotoja [2]. Artikkelissa kriittisellä paikalla tarkoitetaan sellaista kohtaa ohjelmakoodissa, joka on haavoittuvainen tietyntyyppisille hyökkäyksille ja johon verkkohyökkäys voi kohdistua.

WWW-sovellusten haavoittuvuuksia voi testata kahdella pääasiallisella tavalla: *White-box*-analyysillä ja *black-box*-testauksella [2]. White-box-analyysi tarkoittaa sovelluksen lähdekoodin tarkastelua joko manuaalisesti tai automaattisesti suorittamatta sitä, kun taas black-box-testauksessa sovellusta ajetaan ja verrataan sen antamaa tulostetta odotettuun arvoon tietyillä syötteillä. Molemmissa tavoissa on heikkoutensa: kompleksista koodia voi olla vaikea analysoida, ja black-box testaus tarkastelee vain sovelluksen tulostetta jättäen sovelluksen sisäisen tilan huomiotta [2].

Kaksi viimeistä artikkelia käsittelevät CSRF-hyökkäyksiä. Zeller ja Felten [20] antavat esimerkkejä todellisista CSRF-haavoittuvuuksista ja antavat suosituksia siitä, kuinka WWW-palvelimet voivat torjua CSRF-hyökkäyksiä. Lisäksi artikkelissa kerrotaan, miten CSRF-suojaus tulisi toteuttaa siten, ettei se tuota ongelmia tavanomaiselle WWW-selailulle. Aikaisemmat CSRF-suojaukset ovat esimerkiksi vaatineet palvelimen tallentavan tietoja CSRF-suojaukseen liittyen tai tehneet useamman välilehden käytön mahdottomaksi [20]. Lopuksi he ehdottavat, kuinka selaimet itsessään voivat ehkäistä CSRF-hyökkäyksen tapahtumista sekä esittävät tätä varten toteuttamansa Firefox-liitännäisen.

Barth, Jackson ja Mitchell [5] esittelevät *Login CSRF*-haavoittuvuuden, joka on saanut verrattain vähän huomiota, vaikka sillä voi olla yhtä vakavia seurauksia kuin XSS-hyökkäyksellä [5]. Login CSRF tapahtuu, kun hyökkääjä kirjautuu (omilla tunnuksillaan) jollekin sivustolle käyttäen käyttäjän selainta. Jos kyseessä olisi esimerkiksi hakukone, hyökkääjä voisi vakoilla käyttäjän hakuhistoriaa [5]. Artikkelissa kerrotaan myös kolme tapaa torjua CSRF-hyökkäyksiä, joista jokaisen vahvuudet ja heikkoudet punnitaan. Näitä tapoja ovat salaisen tiivistearvon käyttäminen, HTTP Referer-otsikkotiedon hyödyntäminen sekä mukautetun HTTP-otsikkotiedon lähettäminen. He kuitenkin ehdottavat pitkän tähtäimen ratkaisuksi *Origin*-otsikkotietoa, jonka selaimen kuuluisi lähettää tietona siitä, mistä lähteestä HTTP-pyyntö on peräisin.

3 Peruskäsitteitä

Luvussa kuvataan lyhyesti PHP-ohjelmointikieli ja SQL-kieli, joita käytetään myöhemmissä luvuissa esimerkkikoodien yhteydessä.

3.1 PHP

PHP (akronyymi termistä *PHP: Hypertext Preprocessor*) on vuonna 1995 ilmestynyt ja laajalti käytössä oleva yleiskäyttöinen ohjelmointikieli, jota käytetään erityisesti

dynaamisten WWW-sovellusten kehityksessä [9]. PHP toimii siten, että WWW-palvelin prosessoi PHP-tulkin avulla PHP-tiedostossa olevaa PHP-koodia, minkä lopputuloksena syntyy selaimelle lähetettävä WWW-sivu.

PHP:hen on mahdollista asentaa laajennuksia, jotka lisäävät sen toiminnallisuutta. MySQLi on eräs sisäänrakennettu laajennus, joka tarjoaa rajapinnan MySQL-relaatiotietokannanhallintajärjestelmää varten.

Tässä tutkielmassa käytetään juuri MySQLi-rajapintaa tietokannan kanssa kommunikointiin.

3.2 SQL

SQL (Structured Query Language) on tekstimuotoinen kieli, jota käytetään kommunikointiin relaatiotietokantojen kanssa [1]. Yhdellä SQL-lauseella voi esimerkiksi hakea, poistaa, päivittää tai lisätä tietoja tietokantaan. Lauseeseen voidaan lisätä ehtoja, joilla rajoitetaan kysely koskemaan vain tiettyjä rivejä. Esimerkiksi seuraavalla SQL-lauseella voidaan hakea `users`-taulusta niiden henkilöiden tunnisteet ja nimet, joiden nimi alkaa A-kirjaimella:

```
SELECT uid, name FROM users WHERE name LIKE 'A%';
```

WWW-sovelluksissa on tavallista, että SQL-lauseen ehtolausekkeen parametri on käyttäjän syöttämä. Jos tietokantaan syötettävää SQL-lausetta ei konstruoida huolellisesti, sovellukseen saattaa jäädä SQL-injektiohaavoittuvuus.

4 SQL-injektio

SQL-injektio tarkoittaa sitä, että hyökkääjä muuttaa WWW-sovelluksessa olevan SQL-kyselyn rakennetta lisäämällä kyselyyn omia SQL-avainsanoja tai operaattoreita [11]. Listauksessa 1 on esimerkki SQL-injektiohaavoittuvuudesta PHP-koodissa. Haavoittuvuus johtuu siitä, että käyttäjän syöttämä `username`-parametri liitetään sellaisenaan osaksi SQL-kyselyä.

Listaus 1: Esimerkki SQL-injektiolle haavoittuvaisesta PHP-koodista.

```
$username = $_POST["username"];
$query = "SELECT name FROM users WHERE name LIKE '$username'";
$result = $mysqli->query($query);
```

Hyökkääjä voisi käyttää tätä haavoittuvuutta hyväkseen syöttämällä `username`-parametrin arvoksi esimerkiksi `' OR 1=1 --`. Tällä tavalla hän voisi saada selville kaikkien tietokannassa olevien käyttäjien tunnukset.

4.1 Injektiomekanismit

On olemassa useita mekanismeja, joiden kautta SQL-injektio tapahtuu. Näistä yleisimpiä ovat käyttäjän syöte, evästeet sekä palvelinmuuttujat. [11] Listauksessa 1 mainittu tapaus on esimerkki käyttäjän syötteeseen perustuvasta injektiohaavoittuvuudesta. Useimmissa SQL-injektiohyökkäyksissä käyttäjän syötteet ovat peräisin WWW-sivulla olevasta lomake-elementistä [11].

Käyttäjän syöte ei kuitenkaan ole ainoa keino aiheuttaa SQL-injektiota. Selain lähettää HTTP-pyynnössä palvelimelle tavallisesti käyttäjän syötteen lisäksi myös sivustoon liittyvät evästeet. Jos WWW-sovellus käyttää evästeitä SQL-kyselyjen luomisessa, hyökkääjä voisi suorittaa SQL-injektion evästeen avulla [11]. Jos edellisessä esimerkissä `$_POST`-muuttujan paikalla olisi muuttuja `$_COOKIE`, olisi kyseessä evästeeseen perustuva haavoittuvuus.

Kolmas SQL-injektiomekanismi on palvelinmuuttujien hyödyntäminen. PHP:ssä palvelinmuuttujat ovat WWW-palvelimen itsensä generoimia, mutta ne voivat silti sisältää tietoja, jotka ovat lähtöisin käyttäjältä. Tällaisia ovat esimerkiksi muuttujat `$_SERVER['QUERY_STRING']` sekä `$_SERVER['REQUEST_URI']`. Jos WWW-sovellus tallentaa palvelinmuuttujia sellaisenaan tietokantaan, se voi olla haavoittuvainen SQL-injektiolle [11].

SQL-injektiot voidaan luokitella karkeasti kahteen luokkaan sen mukaan, milloin vahinko tapahtuu. Ensimmäisen asteen injektiossa vahinko tapahtuu heti, kun hyökkääjän syöte saavuttaa tietokannan. Listauksessa 1 on esimerkki tällaisesta injektiosta. Toisen asteen SQL-injektiossa käyttäjän syöte tallennetaan aluksi turvallisesti koodattuna tietokantaan, mutta sitä käytetään toisessa SQL-kyselyssä myöhemmin, jolloin varsinainen vahinko tapahtuu [11].

Listaus 2: Esimerkki toisen asteen SQL-injektiosta.

```
1 //Oletetaan, että $uid on käyttäjän tunniste.
2 $item = $mysqli->real_escape_string($_POST["item"]);
3 $mysqli->query("INSERT INTO items VALUES($uid, '$item')");
4 //Myöhemmin koodissa:
5 $result = $mysqli->query("SELECT item FROM items WHERE uid = $uid");
6 $row = $mysqli->fetch_array($result);
7 $item = $row[0];
8 $mysqli->query("SELECT uid FROM items WHERE item LIKE '$item'")
```

Listauksessa 2 on esimerkki toisen asteen SQL-injektiosta. Rivillä 2 käyttäjän syöttämä parametri `item` koodataan (*escape*). Tämä tarkoittaa sitä, että syötteessä olevat erikoismerkit pakotetaan tulkittavaksi tavallisina merkkeinä. Tämän jälkeen syöte tallennetaan tietokantaan rivillä 3. Kyseinen tavara haetaan myöhemmin tietokan-

nasta riveillä 5–6, ja se tallennetaan muuttujaan `$item` rivillä 7. Injektiohaavoittuvuus syntyy siitä, että muuttujaa `$item` ei ole koodattu, ja se liitetään sellaisenaan osaksi uutta SQL-kyselyä rivillä 8.

4.2 Injektiotyyppejä

SQL-injektiot voidaan jakaa tiettyihin alatyyppeihin sen mukaan, mitä injektiolla halutaan saada aikaan. Usein eri tyyppejä ei käytetä yksikseen vaan yhtä aikaa tai peräkkäin [11]. Yleisimmät tyypit ovat tautologiaan perustuva hyökkäys, UNION-hyökkäys ja reppuselkäkysely.

Tautologiaan perustuvassa hyökkäyksessä ehdolliseen SQL-lauseeseen liitetään sellainen osa, joka tekee ehtolausekkeesta aina toden [11]. Listauksen 1 yhteydessä esitetty esimerkki hyökkäyksestä on tällainen, sillä lopulliseksi SQL-lauseeksi muodostuisi

```
SELECT name FROM users WHERE name LIKE '' OR 1=1 --'
```

joka on tautologian vuoksi yhtäpitävä lauseen `SELECT name FROM users` kanssa.

UNION-hyökkäyksessä SQL-kyselyyn liitetään osa, jolla tulosjoukkoon saadaan liitettyä tietoa toisesta tietokannan taulusta [11]. Hyökkäys edellyttää sitä, että hyökkääjä tietää kyseisen tietokannan taulun ja sen sarakkeiden nimet.

Reppuselkäkyselyt (*Piggy-Backed queries*) ovat injektioita, jotka sisältävät uuden SQL-lauseen puolipisteellä erotettuna [11]. Esimerkiksi listauksessa 1 hyökkääjä voisi syöttää `username`-parametrina merkkijonon `';DROP TABLE users--`, jolloin muuttujaan `$query` tallentuisi kaksi eri SQL-lausetta. Jälkimmäinen lause tuhoaisi tietokannassa olevan koko `users`-taulun sisällön.

4.3 SQL-injektioiden ehkäiseminen

SQL-injektiohaavoittuvuudet johtuvat useimmiten siitä, etteivät WWW-sovelluskehittäjät ole noudattaneet turvallisia ohjelmointikäytänteitä [16]. Injektioiden ehkäisemiseksi on olemassa kaksi tapaa: parametrisoidut kyselyt sekä syötteen koodaaminen.

Parametrisoitu kysely tarkoittaa keinoa erottaa SQL-kyselyn ”muotti” sekä siihen liittyvät parametrit toisistaan. Kyseiset parametrit voivat olla esimerkiksi käyttäjän syöttämiä.

Listaus 3: Esimerkki parametrisoitujen kyselyjen käytöstä.

```
1 $username = $_POST["username"];
2 $stmt = $mysqli->prepare("SELECT name FROM users WHERE name LIKE ?");
```

```
3 $stmt->bind_param('s', $username);
4 $stmt->execute();
```

Listauksessa 3 on listauksen 1 kysely parametrisoidussa muodossa. Kyselyn muotti (*prepared statement*) luodaan rivillä 2, ja siihen liitetään `$username`-parametri rivillä 3. Kutsun `bind_param` ensimmäinen parametri `'s'` kertoo parametrin tyypin, joka tässä tapauksessa on merkkijono. Varsinainen kysely suoritetaan rivin 4 `execute`-kutsulla.

Parametrisoidun SQL-kyselyn rakenne on ohjelmoijan ennalta määräämä, jolloin minkäänlainen SQL-injektio ei ole mahdollinen [16]. Ohjelmoijan ei siis tarvitse itse huolehtia käyttäjän syöttämien parametrien koodauksesta, vaan se tapahtuu automaattisesti.

Hyvin vanhoissa PHP-sovelluksissa ei kuitenkaan ole mahdollista käyttää parametrisoituja kyselyitä, sillä MySQLi-rajapinnan käyttö on mahdollista vasta MySQL:n versiosta 4.1.3 alkaen [10]. Silloin ohjelmoijan on luotava SQL-kysely dynaamisesti liittäen merkkijonoja yhteen. Esimerkiksi listauksen 1 SQL-lause

```
$query = "SELECT name FROM users WHERE name LIKE '$username'";
```

on sama kuin

```
$query = "SELECT name FROM users WHERE name LIKE '" . $username . "'";
```

Kyseisen SQL-kyselyn lopullinen rakenne määräytyy siitä, mitä parametri `$username` sisältää. Siksi ohjelmoijan on itse huolehdittava siitä, ettei kyseinen parametri sisällä mitään sellaisia erikoismerkkejä, jotka voisivat muuttaa SQL-lauseen rakennetta.

Dynaamisten SQL-kyselyjen yhteydessä paras tapa välttää SQL-injektio on koodata käyttäjän syötteet [16]. Vanhoissa PHP-sovelluksissa tämä onnistuu listauksessa 1 esitellyllä funktiolla `real_escape_string`. Ohjelmoijan on itse muistettava koodata kaikki käyttäjän syötteet käyttäen ohjelmointiympäristön tarjoamia koodausmenetelmiä [16]. Omatekoisten koodausmenetelmien käyttäminen ei luonnollisesti ole suositeltavaa, sillä niihin voi jäädä ohjelmointivirheitä.

5 Cross-Site Scripting

Cross-Site Scripting (XSS) on SQL-injektion tapaan koodi-injektioihin kuuluva haavoittuvuus. XSS-hyökkäyksessä WWW-sovellus tulostaa käyttäjän selaimeen lähetettävälle WWW-sivulle hyökkääjän koodia, jonka selain suorittaa saadessaan WWW-sivun palvelimelta. SQL-injektioiden tapaan XSS-haavoittuvuudet johtuvat käyttäjän syötteen riittämättömästä tarkistamisesta. [17]

Listaus 4: Yksinkertainen esimerkki XSS-haavoittuvuudesta.

```
echo "<div>Hello, " . $_GET["name"] . "!</div>";
```

Listauksessa 4 on esimerkki PHP-koodista, jonka on tarkoitus tulostaa selaimeen tervehdys annettuun nimeen perustuen. Jos hyökkääjä nyt antaisi `name`-parametrin arvoksi esimerkiksi merkkijonon

```
<a href="http://site.com/">Test</a>
```

niin WWW-sivulle tulostuisi hyperlinkki ulkopuoliselle sivustolle, joka voisi olla haitallinen.

Nykyiset WWW-selaimet noudattavat ns. saman lähteen käytäntöä (*same origin policy*). Kaksi lähdettä määritellään samoiksi, kun niillä on sama isäntänimi (*hostname*), portti sekä protokolla [4]. Kyseinen käytäntö rajoittaa lähdettä A lukemasta dataa eri lähteestä B tiettyjä poikkeuksia lukuun ottamatta [4].

WWW-sivulle on mahdollista upottaa JavaScript-koodia, jota tavallisesti käytetään tekemään WWW-sovelluksista interaktiivisempia. Tällaisella koodilla on pääsy kaikkiin sellaisiin objekteihin, joilla on sama lähde. Näitä ovat esimerkiksi sivuston evästeet sekä WWW-sivu itse [12]. WWW-palvelimet käyttävät evästeitä säilyttämään käyttäjän tilan sivustolla [3].

XSS-injektiossa voidaan käyttää hyväksi saman lähteen käytäntöä [12], koska hyökkääjän injektoima koodi on samalla sivulla. Hyökkääjä voi esimerkiksi injektoida WWW-sivulle sinne kuulumatonta JavaScript-koodia käyttäen `<script>`-tagia. Näin on mahdollista varastaa käyttäjän evästeet, vakoilla käyttäjää tai kaapata käyttäjätili [12, 17].

5.1 Injektiotyyppejä

XSS-injektiot voidaan jakaa kolmeen eri tyyppiin sen mukaan, millaisessa yhteydessä injektio tapahtuu. Kyseiset tyypit ovat pysyvä XSS (*stored XSS*), heijastettu XSS (*reflected XSS*) sekä DOM-pohjainen XSS (*DOM-based XSS*) [17].

Heijastetussa XSS-hyökkäyksessä WWW-palvelin sisällyttää HTTP-pyynnössä käyttäjän antamaa syötettä WWW-sivuun, joka lähetetään HTTP-vastauksessa selaimelle [17]. Listauksessa 4 on esimerkki tällaisesta tapauksesta. Injektio tapahtuu siis vain silloin, kun WWW-sivulle syötetään haitallisia parametreja. Käyttäjä voisi esimerkiksi altistua hyökkäykselle painamalla hyperlinkkiä, jonka `href`-attribuuttina on seuraava:

```
"http://site.com/?name=<script>alert('XSS')</script>"
```

Pysyvä XSS-injektio tapahtuu, kun WWW-palvelin tallentaa haitallista syötettä tietokantaan ja myöhemmin esittää tämän syötteen WWW-sivulla [17]. Haittakoodi

on siis tallennettu pysyvästi WWW-sovellukseen, toisin kuin heijastetussa injektiossa. Tällöin käyttäjä altistuu haittakoodille vieraillessaan sivulla riippumatta annetusta syötteestä.

DOM-pohjainen haavoittuvuus johtuu siitä, että WWW-sivulla olevat komentosarjat käsittelevät käyttäjän syötettä huolimattomasti, mikä johtaa XSS-injektioon [17]. Esimerkiksi seuraava WWW-sivulla oleva komentosarja on altis hyökkäykselle:

```
<script>document.write(document.location.href)</script>
```

Jos hyökkääjä saa uhrin painamaan linkkiä

```
”http://site.com/?<script>alert(’XSS’)</script>”
```

niin silloin selain suorittaa linkkitekstin `<script>`-tagissa olevan komentosarjan, mikäli selain ei automaattisesti koodaa `document.write`-metodin merkkijonoparametria.

5.2 XSS-injektioiden ehkäiseminen

XSS-injektiot voidaan torjua riittävällä syötteen tarkistamisella. Tähän on olemassa neljä perustapaa. Syötteessä olevat erikoismerkit voidaan korvata toisilla merkeillä tai poistaa kokonaan. Erikoismerkit voidaan myös koodata eli ne pakotetaan tulkittavaksi tavallisina merkkeinä. Neljäs keino on määritellä vain sallitut merkit, joita käyttäjä voi WWW-sovellukseen syöttää. [17]

Ongelma erikoismerkkien muokkaamisessa tai poistamisessa on se, että käyttäjän antaman viestin alkuperäinen merkitys voi hämärtyä. Sallittujen merkkien rajoittaminen voi puolestaan johtaa monen käyttäjän syötteen hylkäämiseen [17]. Nämä ongelmat voidaan välttää syötteen koodaamisella.

Yleisessä tapauksessa käyttäjän syötteen koodaaminen XSS-injektion ehkäisemiseksi ei ole yksinkertaista. Syötteen oikeanlainen koodaus tarvitsee tiedon siitä, missä rakenteellisessa kohdassa käyttäjän syöte esiintyy WWW-sivulla [19]. Esimerkiksi listauksessa 4 syöte esiintyy HTML-tagin sisällä. Silloin riittää koodata HTML-erikoismerkit [19]. Toisaalta käyttäjän syöte voisi olla `href`-attribuutin sisällä:

```
echo ’<a href="’ . $_POST[’address’] . ’">Test</a>’;
```

Tällöin HTML-erikoismerkkien koodaus ei ole enää riittävä suojauskeino XSS-injektiota vastaan, sillä se ei riitä suojaamaan muun muassa `javascript:`-alkuisia URI-osoitteita vastaan [19]. Tämä pätee vain siinä tapauksessa, että käyttäjän syöte muodostaa koko linkin. Silloin ennen syötteen koodausta on tarkistettava, että syötetyn linkin protokolla on kelvollinen [15], ts. että se alkaa merkkijonolla `http://`. Merkkijono

```
javascript:alert(document.location)
```

on esimerkki `javascript:`-alkuisesta osoitteesta. Tällaista linkkiä painettaessa selain näyttäisi käyttäjälle ponnahdusikkunan, jossa tekstinä olisi sivuston osoite.

Toinen ongelma XSS-injektion ehkäisemisessä on se, että käyttäjän syöte voi olla kahdessa erilaisessa sisäkkäisessä rakenteessa [19]. Listauksessa 5 on esimerkki tällaisesta tilanteesta, jossa syöte on toisaalta heittomerkkien sisällä JavaScript-koodissa, mutta myös `script`-tagien välissä.

Listaus 5: Esimerkki käyttäjän syötteestä sisäkkäisessä rakenteessa.

```
echo "<script> var address = '" . $POST_['address'] . "'</script>";
```

Tässä tilanteessa hyökkääjä voisi käyttää syötteessään joko heittomerkkiä tai lopettavaa `</script>`-tagia aiheuttaakseen injektion [19].

On olemassa myös useita muita rakenteita, joissa käyttäjän syöte tarvitsee tietynlaisen koodauksen [15], mutta nämä tapaukset ovat kuitenkin harvinaisia [14]. Useimmissa tapauksissa riittää noudattaa seuraavaa kahta sääntöä [15]:

- Käyttäjän syötettä ei koskaan pidä sisällyttää `script`- eikä `style`-tagien väliin.
- Käyttäjän syötteen saa sisällyttää vain HTML-elementin sisään (ei siis attribuutin arvoksi eikä nimeksi), ja silloin sen sisältämät HTML-erikoismerkit on koodattava.

PHP:ssä HTML-erikoismerkit voidaan koodata `htmlspecialchars`-funktiolla [8]. Tätä voidaan soveltaa esimerkiksi listauksen 4 tapaukseen seuraavasti:

```
echo "<div>Hello, " . htmlspecialchars($_GET["name"]) . "!</div>";
```

Tämä ehkäisee XSS-injektiot, sillä se täyttää edellä mainitut kaksi sääntöä.

6 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF)-hyökkäyksessä käyttäjän selain saadaan lähettämään käyttäjän tietämättä HTTP-pyyntö toiselle sivustolle [20]. CSRF voi tapahtua käyttäjän vieraillessa haitallisella sivustolla, joka käskee selainta tekemään pyynnön toiselle sivustolle [5]. Hyökkäyksen onnistuminen johtuu siitä, ettei pyyntöä vastaanottava sivusto tarkista, mistä lähteestä pyyntö on peräisin. Sivusto siis luulee, että pyyntö lähetettiin käyttäjän omasta toimesta [20].

Monet WWW-sivustot ovat haavoittuvia CSRF-hyökkäyksille, vaikka niiden torjuminen on helppoa. Haavoittuvuudet johtuvat siitä, että sovelluskehittäjät ovat tietämättömiä CSRF:n syistä ja seurauksista. [20] Lisäksi saman lähteen käytäntö ei anna suojaa CSRF-hyökkäyksiä vastaan, koska se ei kiellä eri lähteitä lähettämästä pyyntöjä toisilleen [20].

6.1 Esimerkki CSRF-haavoittuvuudesta

Oletetaan, että käyttäjän tuntemalla sivustolla listauksessa 6 oleva lomake.

Listaus 6: Esimerkki WWW-sivulla olevasta lomakkeesta.

```
<form action="http://site.com/sendmsg.php" method="GET">
Message: <input type="text" name="msg" />
<input type="submit" value="Send message" />
</form>
```

Käyttäjän painaessa `Send message`-painiketta selain lähettää HTTP-pyynnön osoitteeseen `http://site.com/sendmsg.php?msg=xxx`, jossa `xxx` on käyttäjän antama syöte. Sama pyyntö voitaisiin kuitenkin lähettää kirjoittamalla kyseinen osoite suoraan selaimen osoitepalkkiin, eikä WWW-palvelin huomaisi pyynnössä eroa. Hyökkääjä voisi hyödyntää tätä seikkaa laittamalla sivustolleen linkin kyseiseen osoitteeseen. Linkki voisi olla esimerkiksi `<img>`-tagin `src`-attribuutissa. Silloin käyttäjän riittäisi vierailla hyökkäyssivustolla, jolloin selain lähettäisi pyynnön sivustolle yrittäessään ladata kuvaa [20].

CSRF-haavoittuvuus on sitä vakavampi, mitä enemmän käyttäjällä on oikeuksia sivustolla [20]. Tämä johtuu siitä, että HTTP-pyynnössä sivustolle lähetetään sivustoon liittyvät evästeet, vaikka pyyntö olisi lähtöisin hyökkäyssivustolta. Hyökkääjä pystyy siis tekemään toimintoja sivustolla aivan kuin sisään kirjautunut käyttäjä.

6.2 CSRF-haavoittuvuuksien ehkäiseminen

CSRF-hyökkäyksiä vastaan on olemassa useita suhteellisen yksinkertaisia keinoja, jotka joko torjuvat ne kokonaan tai osittain.

Ensimmäinen keino on varmistaa, että GET-tyyppiset HTTP-pyynnöt voivat ainoastaan hakea sivustolta tietoa eikä muokata sitä [20]. Tämä sääntö noudattaa myös HTTP 1.1-protokollaa, jonka mukaan GET-metodin tulisi ainoastaan noutaa tietoa palvelimelta [7]. Edellisessä esimerkissä tätä sääntöä ei ole noudatettu: lomakkeen lähetysmetodiksi on merkitty GET. Jos metodiksi vaihdettaisiin POST, esimerkissä mainittu hyökkäys ei olisi mahdollinen, sillä selain käyttää kuvien lataamiseen aina

GET-metodia.

Edellinen keino ei kuitenkaan ole riittävä, sillä esimerkiksi JavaScript-koodin avulla on mahdollista lähettää HTTP-pyyntöjä myös POST-metodilla. Zeller ja Felten [20] ehdottavat, että jokaisen POST-pyynnön mukana tulee lähettää satunnaisluku, joka on vaikeasti arvattavissa. Tämän satunnaisluvun täytyy olla sama kuin sivuston evästeessä oleva. Koska hyökkääjä ei voi lukea käyttäjän evästeitä saman lähteen käytännöstä johtuen, CSRF-hyökkäys toteutuu vain, jos hyökkääjä onnistuu arvaamaan kyseisen satunnaisluvun. [20]

Generoidun satunnaisluvun arvo ei saa riippua käyttäjätilistä, jotta se torjuisi myös ns. sisäänkirjautumis-CSRF-hyökkäyksen (*Login CSRF*) [20]. Tällaisessa hyökkäyksessä hyökkääjä saa käyttäjän kirjautumaan sisään WWW-palveluun hyökkääjän omilla tunnuksilla. Tällä tavalla hyökkääjä voi esimerkiksi vakoilla käyttäjän toimia kyseisellä sivustolla. [5]

Jos satunnaisluku riippuisi käyttäjätilistä (esimerkiksi käyttäjätunnuksesta ja/tai salasanasta), hyökkääjä voisi katsoa oman satunnaislukunsa ja siten liittää kyseisen luvun hyökkäyssivuston lomakkeeseen. Käyttäjän joutuessa hyökkäyssivustolle Login CSRF-hyökkäys onnistuisi.

Vaihtoehtoinen tapa CSRF-hyökkäyksen torjumiseksi on tarkistaa HTTP-pyynnön Referer-otsikkotieto [5]. Tämä otsikkotieto kertoo, miltä WWW-sivulta pyyntö on peräisin [7]. Tällöin CSRF-hyökkäyksen torjumiseksi riittää tarkistaa, että pyyntö on lähtöisin samalta sivustolta kuin johon pyyntö kohdistuu [5].

Referer-otsikkotiedon käyttäminen CSRF-hyökkäyksen torjumiseksi ei kuitenkaan ole täysin ongelmatonta. Kyseinen otsikkotieto voi toisinaan vuotaa käyttäjästä tietoa [13]. Siksi jotkut internetin käyttäjät ovat estäneet sen lähetyksen HTTP-pyynnön mukana [5]. Otsikkotiedon puuttuessa sivusto ei voi enää päätellä pyynnön alkuperää, joten se joutuu näyttämään virheilmoituksen.

6.3 Toteutus PHP-sovelluksissa

PHP:ssä CSRF-suojaus voidaan toteuttaa käyttäen `$_SESSION`-muuttujaa, jonne tallennetaan käyttäjän istuntoon liittyviä tietoja. Yksi näistä tulee olemaan CSRF-suojauksessa käytettävä satunnaisluku.

Suojataan listauksen 6 esimerkin lomake CSRF-haavoittuvuudelta. Tämä on toteutettu listauksessa 7. Rivillä 1 tarkistetaan, onko käyttäjä lähettämässä viestiä POST-pyynnöllä. Jos ei ole, eli käyttäjä tulee lomakesivulle ensimmäistä kertaa, niin riveillä 4–8 tulostetaan WWW-sivulle lomake, jolla viestin voi lähettää. Tätä ennen arvotaan satunnaisarvo rivillä 2 ja tallennetaan se käyttäjän istuntoon rivillä 3. Satunnaisarvo

liitetään osaksi lomaketta rivillä 7.

Kun käyttäjä lähettää lomakkeen, niin `sendmsg.php` suoritetaan uudelleen, jolloin rivin 1 `isset`-funktio palauttaa `true`. Silloin suoritus jatkuu riviltä 10, jossa tarkistetaan, ovatko lomakkeessa oleva arvo ja käyttäjän istuntoon tallennettu arvo samoja. Jos yhtäsuuruus toteutuu, voidaan olla varmoja siitä, että käyttäjä on itse lähettänyt lomakkeen. Muussa tapauksessa kyseessä on CSRF-hyökkäysyritys, jolloin voidaan esimerkiksi tulostaa virheilmoitus.

Listaus 7: CSRF-suojaus `sendmsg.php`-sivulla.

```
1  if (!isset($_POST['msg'])) {
2    $rand = hash("sha512", mt_rand());
3    $_SESSION['csrf'] = $rand;
4    echo '<form action="http://site.com/sendmsg.php" method="POST">';
5    echo 'Message: <input type="text" name="msg" />';
6    echo '<input type="submit" value="Send message" />';
7    echo '<input type="hidden" name="csrf" value="' . $rand . '" />';
8    echo '</form>'
9  } else {
10   if ($_SESSION['csrf'] === $_POST['csrf']) {
11     //Pyyntö on aito. Tallenna lähetetty viesti.
12   } else {
13     //CSRF-hyökkäys. Näytä virheilmoitus.
14   }
15 }
```

7 Yhteenveto

XSS-, CSRF- ja SQL-injektiohaavoittuvuudet ovat valitettavan yleisiä WWW-sovelluksissa. Niitä hyväksikäyttävät verkkorikolliset voivat tehdä vakavia vahinkoja yrityksille ja niiden asiakkaille. Tutkielmassa tarkasteltiin, kuinka näitä haavoittuvuuksia voidaan ehkäistä PHP-sovelluksissa defensiivisillä ohjelmointimenetelmillä.

SQL-injektion torjuminen on nykyään yksinkertaista parametrisoitujen kyselyjen ansiosta. Niitä käyttäen ohjelmoijan ei itse tarvitse muistaa käsitellä käyttäjän syötettä, koska SQL-kyselyn rakenne on täysin ohjelmoijan määräämä. Toisen torjuntakeinon eli syötteen koodaamisen ongelmana on se, että ohjelmoijan on itse muistettava koodata käyttäjän syöte silloin, kun se on välttämätöntä. Yksikin unohdus voi altistaa WWW-sovelluksen SQL-injektiolle.

XSS-hyökkäyksen torjuminen yleisessä tapauksessa on huomattavasti vaikeampaa, koska WWW-sivu voi koostua monesta elementistä, joiden sisäinen syntaksi

on erilainen ja siten syötteen koodaamiseen ei voi käyttää yhtä ja samaa funktiota. Useimmissa tapauksissa käyttäjän syöte tulostetaan kuitenkin HTML-elementin sisällä, jolloin syötteen koodaaminen on helppoa.

CSRF-hyökkäysten torjuminen onnistuu suhteellisen helposti. Riittää, että GET-pyyntöjä käytetään ainoastaan tietojen hakemiseen, ja POST-pyyntöjen mukana lähetetään satunnaisarvo. Vaihtoehtoinen tapa torjua CSRF on tarkistaa Referer-otsikkotieto, mutta yksityisyysongelmista johtuen se ei toimi kaikkien käyttäjien kohdalla.

Edellä mainittujen haavoittuvuuksien ehkäiseminen ei ole teknisesti vaikeaa. Kyseiset WWW-sovellusten haavoittuvuudet johtunevatkin yleensä ohjelmoijien tietämättömyydestä tai huolimattomuudesta eivätkä puutteellisista ohjelmointitaidoista.

Lähteet

[1] C. Anley. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, 2002.

[2] N. Antunes ja M. Vieira. Defending against web application vulnerabilities. *Computer*, 45(2):66–72, 2012.

[3] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), 2011.

[4] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), 2011.

[5] Adam Barth, Collin Jackson, ja John C. Mitchell. Robust defenses for cross-site request forgery. Kirjassa *Proceedings of the 15th ACM conference on Computer and communications security*, sarjassa *CCS '08*, CCS '08, ss. 75–88, New York, NY, USA, 2008. ACM.

[6] Steve Christey ja Robert M. Martin. Vulnerability Type Distributions in CVE. `http://cwe.mitre.org/documents/vuln-trends/index.html`, 2007. Verkossa; viitattu 21. syyskuuta 2012.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, ja T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), 1999.

[8] The PHP Group. PHP: htmlspecialchars - Manual. http://php.net/manual/en/function.htmlspecialchars.php, 2012. Verkossa; viitattu 7. marraskuuta 2012.

[9] The PHP Group. PHP: Hypertext Preprocessor. http://www.php.net, 2012. Verkossa; viitattu 26. lokakuuta 2012.

[10] The PHP Group. PHP: MySQLi overview. http://php.net/manual/en/mysqli.overview.php, 2012. Verkossa; viitattu 7. marraskuuta 2012.

[11] William G.J. Halfond, Jeremy Viegas, ja Alessandro Orso. A classification of SQL-Injection attacks and countermeasures. Kirjassa *Proceedings of the IEEE International Symposium on Secure Software Engineering*, ss. 65–81, Arlington, VA, USA, 2006.

[12] Florian Kerschbaum. Simple cross-site attack prevention. Kirjassa *3rd International Conference on Security and Privacy in Communication Networks*, ss. 464–472, 2007.

[13] B. Krishnamurthy ja C.E. Wills. On the leakage of personally identifiable information via online social networks. Kirjassa *Proceedings of the 2nd ACM workshop on Online social networks*, ss. 7–12. ACM, 2009.

[14] OWASP. PHP Security Cheat Sheet. https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet, 2012. Verkossa; viitattu 6. marraskuuta 2012.

[15] OWASP. XSS (Cross Site Scripting) Prevention Cheat Sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, 2012. Verkossa; viitattu 7. marraskuuta 2012.

[16] Lwin Khin Shar ja Hee Beng Kuan Tan. Defeating SQL Injection. Preprint, 2012.

[17] Lwin Khin Shar ja Hee Beng Kuan Tan. Defending against cross-site scripting attacks. *Computer*, 45(3):55–62, 2012.

[18] The PHP Group. Usage Stats for April 2007. http://php.net/usage.php, 2007. Verkossa; viitattu 6. lokakuuta 2012.

[19] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, ja D. Song. A systematic analysis of XSS sanitization in web application frameworks. *Computer Security–ESORICS 2011*, ss. 150–171, 2011.

[20] W. Zeller ja E.W. Felten. Cross-site request forgeries: Exploitation and prevention. Tekninen raportti, Princeton University, 2008.