**Mika Lehtinen**

# OWASP Top 10 -haavoittuvuuksien korjaaminen TIM-järjestelmästä

Tietotekniikan pro gradu -tutkielma

27. marraskuuta 2015

Jyväskylän yliopisto

Tietotekniikan laitos

**Tekijä:** Mika Lehtinen

**Yhteystiedot:** mika.k.lehtinen@student.jyu.fi

**Ohjaaja:** Vesa Lappalainen

**Työn nimi:** OWASP Top 10 -haavoittuvuuksien korjaaminen TIM-järjestelmästä

**Title in English:** Fixing OWASP Top 10 web application vulnerabilities in TIM system

**Työ:** Pro gradu -tutkielma

**Suuntautumisvaihtoehto:** Ohjelmistotekniikka

**Sivumäärä:** 103+8

**Tiivistelmä:** Modernit web-sovellukset ovat monimutkaisia, minkä vuoksi ne voivat sisältää erilaisia haavoittuvuuksia. Jyväskylän yliopiston tietotekniikan laitoksella on kehitteillä *The Interactive Material* -järjestelmä, johon tallennetut dokumentit voivat sisältää interaktiivisia komponentteja ja jossa opiskelijat voivat esimerkiksi tehdä luentomonisteeseen omia muistiinpanojaan. Tutkielmassa käydään läpi web-sovellusten yleisimmät haavoittuvuudet OWASP-organisaation Top 10 -listan mukaan ja selvitetään, mitä näistä haavoittuvuuksista TIM-järjestelmän nykyisessä versiossa on ja kuinka ne voidaan järkevästi korjata. Haavoittuvuuskartoituksessa sovelletaan sekä penetraatiotestausta että lähdekoodin systemaattista läpikäyntiä. Tämän konstruktiivisen tutkielman tuloksena saadaan joukko toteutustekniikoita, joiden avulla haavoittuvuudet voidaan välttää.

**Avainsanat:** haavoittuvuus, OWASP, TIM, web-sovellus

**Abstract:** Modern web applications are complex, which is why they may contain various vulnerabilities. A web application called *The Interactive Material* is being developed at the Department of Mathematical Information Technology, University of Jyväskylä. Documents stored in TIM can contain interactive components and students are able, for instance, to make their own notes in the documents. This thesis presents the most common vulnerabilities found in web applications according to OWASP Top 10 and explores which of these vulnerabilities are present in the current version of TIM and how they can efficiently be fixed.

Both penetration testing and systematic source code analysis are applied in the process of finding vulnerabilities. As a result of this constructive study, we obtain a set of implementation techniques, with the help of which the vulnerabilities can be avoided.

**Keywords:** OWASP, TIM, vulnerability, web application

# Termiluettelo

| | |
|---|---|
| AngularJS | on JavaScript-kehys, joka perustuu MVC-suunnittelumalliin. |
| CSRF | (*Cross-Site Request Forgery*) on web-sovellusten haavoittuvuus. |
| Docker | on avoin alusta, jolla avulla sovellusta voidaan ajaa eristetyssä ympäristössä. |
| DOM | (*Document Object Model*) on tapa esittää HTML-dokumentti sekä vuorovaikuttaa siinä olevien elementtien kesken. |
| Eskapointi | on kontekstista riippuva operaatio, jossa merkkijono muunnetaan sen tietosisältöä muuttamatta sellaiseen muotoon, joka voidaan liittää merkkijono-operaatiolla osaksi rakenteista kyselyä tai dokumenttia muuttamatta kyseisen kyselyn tai dokumentin rakennetta. |
| Flask | on web-ohjelmistokehys. |
| HTML | (*Hypertext Markup Language*) on standardoitu merkintäkieli WWW-sivujen laatimiseen. |
| HTTP | (*Hypertext Transfer Protocol*) on WWW:n käyttämä tiedonsiirtoprotokolla. |
| Komponentti | on ohjelmakirjasto. |
| Korppi-käyttäjä | on käyttäjä, joka on rekisteröitynyt TIM-sovellukseen Korppi-tunnuksen avulla. |
| Ohjelmakirjasto | on sovelluksen käytettävissä oleva kokoelma tietoa ja ohjelmakoodia, joka muodostaa jonkin loogisen kokonaisuuden. |
| OWASP | (*Open Web Application Security Project*) on web-sovellusten tietoturvaan omistautunut yhteisö. |
| SQL | (*Structured Query Language*) on standardoitu tietokantakieli. |
| Sähköpostikäyttäjä | on käyttäjä, joka on rekisteröitynyt TIM-sovellukseen sähköpostin avulla. |
| TIM | (*The Interactive Material*) on Jyväskylän yliopiston tietotekniikan laitoksen e-oppimateriaaliprojekti. |

| | |
|---|---|
| TIM-dokumentti | on TIM-järjestelmään Pandoc-merkintäkielellä tallennettu dokumentti. |
| TIM-järjestelmä | on tim-beta-virtuaalikoneen ja siihen asennettujen ohjelmistojen muodostama kokonaisuus. |
| TIM-sovellus | on Docker-säiliössä ajettava Flask-web-sovellus, joka on TIM-järjestelmän ydinosa. |
| TLS | (*Transport Layer Security*) on protokolla, jolla voidaan suojata tietoliikenne tietoverkossa. |
| URL | (*Uniform Resource Locator*) on viite verkkoresurssiin, kuten WWW-sivuun. |
| Web | on synonyymi termille WWW. |
| WWW | (*World Wide Web*) on Internetissä toimiva palvelujärjestelmä, jonka avulla julkaistaan verkkosivuja ja hyödynnetään niitä. |
| XSS | (*Cross-Site Scripting*) on yleinen web-sovellusten haavoittuvuus. |

# Kuviot



# Taulukot

# Sisältö

# 1 Johdanto

Web-sovellukset ovat vuosien varrella kehittyneet staattisista, yksinkertaisista sivuista dynaamisempaan suuntaan (Jazayeri 2007). Internetin käyttäjä voi nykyään helposti esimerkiksi perustaa oman blogin, kommentoida uutisia, selata karttoja ja jopa suorittaa haluamansa ohjelmointikielen koodia omassa selaimessaan eri web-sovellusten avulla. Myös yritykset ovat ottaneet käyttöönsä web-sovelluksia liiketoimintansa tueksi (Stuttard ja Pinto 2011, 4).

Koska web-sovelluksen ympäristö muodostuu useista eri komponenteista ja tekniikoista, turvallisen web-sovelluksen kehittäminen on haastavaa (Li ja Xue 2011). Monissa web-sovelluksissa onkin *haavoittuvuuksia*, joita hyväksikäyttämällä verkkorikolliset saattavat esimerkiksi varastaa web-sovelluksen sisältämät tiedot. Useita tietomurtoja onkin tapahtunut juuri web-sovellusten tietoturvaongelmien vuoksi (Fonseca ym. 2014). Esimerkiksi Symantecin tutkimista web-sivustoista jopa 77 prosenttia sisälsi ainakin yhden haavoittuvuuden (Wood ym. 2014).

Jyväskylän yliopiston tietotekniikan laitoksella on kehitteillä *The Interactive Material* -järjestelmä, johon tallennetut dokumentit voivat sisältää interaktiivisia komponentteja ja jossa opiskelijat voivat esimerkiksi tehdä luentomonisteeseen omia muistiinpanojaan. Projektin tarkoituksena on luoda malli interaktiivisille ja hallituille e-oppimateriaaleille. Yksi olennaisimmista projektin onnistumiskriteereistä onkin se, ettei kyseinen järjestelmä sisällä haavoittuvuuksia.

Tutkielmassa kartoitetaan, mitä OWASP Top 10 -listan mukaan yleisimpiä web-sovellusten haavoittuvuuksia TIM-järjestelmässä on olemassa ja miten ne voidaan järkevästi korjata. Tämän konstruktiivisen tutkielman tuloksena saadaan joukko toteutustekniikoita, joiden avulla haavoittuvuudet voidaan välttää. Erityisesti tarkastellaan, kuinka TIM-järjestelmän käytössä olevia tekniikoita (kuten Flask-kehys ja AngularJS) voidaan hyödyntää haavoittuvuuksien ehkäisemisessä.

Web-sovelluksen haavoittuvuuksien löytämiseksi on olemassa kaksi tapaa: penetraatiotestaus sekä lähdekoodin analysointi (Austin, Holmgreen ja Williams 2013). Koska molemmissa tavoissa on rajoituksensa (Antunes ja Vieira 2012) ja molempien tapojen hyödyntäminen

on tehokkaampaa kuin vain toisen käyttäminen (Finifter ja Wagner 2011), tutkielmassa testataan TIM-sovellusta kummallakin tavalla. Parhaimmatkaan penetraatiotestausohjelmistot eivät kykene löytämään tietyntyyppisiä haavoittuvuuksia (Doupé, Cova ja Vigna 2010), kuten monia käyttäjän tunnistamiseen ja oikeuksien hallintaan liittyviä haavoittuvuuksia (Dukes, Yuan ja Akowuah 2013).

Luvussa 2 määritellään web-sovellus ja kuvataan siihen liittyvät olennaiset käsitteet. Luvussa 3 esitellään web-sovellusten yleisimmät haavoittuvuudet ja niiden ehkäisemiseksi käytetyt keinot. Luvussa 4 kartoitetaan keinoja, joilla haavoittuvuuksia voidaan etsiä web-sovelluksista. Luvussa 5 esitellään TIM-sovelluksen taustaa ja sen rakenne. Luvussa 6 esitellään tutkimusmetodi. Luvussa 7 suoritetaan TIM-sovelluksen black-box-testaus, ja luvussa 8 suoritetaan TIM-sovelluksen white-box-analyysi sekä esitellään löydettyjen haavoittuvuuksien korjauskeinot.

# 2 Web-sovelluksen toiminta

Luvussa määritellään web-sovellus ja kuvataan siihen liittyvät seuraavat olennaiset käsitteet: URL, HTTP-protokolla ja HTML-merkintäkieli.

## 2.1 Web-sovellus ja URL

*Web-sovellus* on sovellus, joka toimii Internetin välityksellä selainta käyttäen (Jazayeri 2007). Web-sovelluksen toimiessa käyttäjän eli asiakkaan selain lähettää pyyntöjä – joko käyttäjän aloitteesta tai automaattisesti – web-sovelluksen palvelimelle, joka vastaa kuhunkin pyyntöön lähettämällä paluuviestin selaimelle. Yksittäinen pyyntö voi olla esimerkiksi etusivun, siinä olevan kuvan tai muun resurssin hakeminen tai lomakkeen tietojen lähettäminen.

Selaimen ja web-palvelimen kommunikointi perustuu URL-osoitteisiin ja HTTP-protokollaan (Zalewski 2012). URL-osoitteen avulla selain voi viitata web-sovelluksen eri osiin, kuten sivuihin, kuviin tai muihin resursseihin. URL-osoite on tavallisesti muotoa `scheme://address:port/path?query#fragment` (Berners-Lee, Fielding ja Masinter 2005), missä eri osat tarkoittavat seuraavia:

- `scheme` on käytettävä protokolla (Web-sovellusten tapauksessa `http`),
- `address` on web-palvelimen osoite,
- `:port` on portin numero, johon palvelimella otetaan yhteyttä (HTTP:ssä oletuksena 80),
- `/path` on polku web-palvelimella sijaitsevaan resurssiin,
- `?query` sisältää parametrit sekä
- `#fragment` on tarkennin selainta varten.

URL-osoitteen osista vain `scheme`, `address` ja `/path` ovat pakollisia.

## 2.2 HTTP-protokolla

Selaimen lähettämä pyyntö web-sovellukselle on HTTP-protokollan mukainen. HTTP-pyyntö on tekstimuotoinen, ja sen osat ovat pyyntörivi (*request line*), otsikkorivit (*headers*) sekä sisältöosa (*body*) (Fielding ja Reschke 2014a).

Listauksessa 2.1 on esimerkki HTTP-pyynnön sisällöstä, jossa ensimmäinen rivi on pyyntörivi ja kaksi seuraavaa ovat otsikkorivejä. Pyynnön lopussa on kaksi tyhjää riviä, mikä tarkoittaa, että sisältöosa on tyhjä. Otsikkoriveistä vain `Host` on pakollinen. Se ilmaisee kohdepalvelimen osoitteen. `Accept`-otsikkorivi ilmoittaa, mitä sisältötyyppejä selain on valmis vastaanottamaan palvelimelta.

Listauksen 2.1 HTTP-pyynnön metodi on GET. Sitä käytetään tiedon hakemiseen palvelimelta (Fielding ja Reschke 2014b). GET-metodin on tarkoitus olla sivuvaikutukseton, ts. sen ei pitäisi muuttaa web-sovelluksen tilaa (Zalewski 2012, 52). Tällöin esimerkiksi selain voi turvallisesti hakea web-sivun linkit ennakkoon. Tallennukseen soveltuvia metodeja ovat POST ja PUT.

```
GET http://www.example.com/ HTTP/1.1
Host: www.example.com
Accept: text/html


```

Listaus 2.1. Esimerkki HTTP-pyynnön sisällöstä.

Listauksessa 2.2 on esimerkki palvelimen HTTP-vastauksesta, joka on rakenteeltaan hyvin samankaltainen kuin HTTP-pyyntö. Ensimmäinen rivi on tilarivi, joka ilmoittaa käytössä olevan protokollan ja sen version sekä tilakoodin (Fielding ja Reschke 2014a). Seuraavat epätyhjät rivit ovat otsikkorivejä, ja vastauksen sisältöosa on edelleen erotettu tyhjällä rivillä (Fielding ja Reschke 2014a). Esimerkin otsikkorivit kertovat, että vastauksen sisältönä on pelkkää tekstiä (`text/plain`) ja että sen pituus on 6 tavua.

```
HTTP/1.1 200 OK
Content-Length: 6
Content-Type: text/plain

Hello!
```

Listaus 2.2. Esimerkki palvelimen HTTP-vastauksesta.

### 2.2.1 HTTPS

Selaimen ja palvelimen välinen HTTP-tietoliikenne on selkokielistä eli salaamatonta. Tästä seuraa, että kolmannen osapuolen – etenkin niiden, joiden kautta tietoliikenne kulkee – on mahdollista salakuunnella ja muuttaa HTTP-pyyntöjen ja -vastausten sisältöä. Tämän ongelman välttämiseksi selain voi muodostaa palvelimeen salatun *HTTPS*-yhteyden, kunhan molemmat osapuolet tukevat sitä. HTTPS-yhteys ilmaistaan URL-osoitteessa `https://`-etuliitteellä.

HTTPS käyttää TLS-protokollaa salatun yhteyden muodostamiseen. HTTPS-yhteyden muodostuksessa palvelin lähettää selaimelle *sertifikaatin*, joka sisältää sivuston identiteetin (verkkotunnuksen), validiusjakson, julkisen avaimen sekä digitaalisen allekirjoituksen. Selain varmistaa, että allekirjoitus on pätevä, sertifikaatin identiteetti vastaa haluttua verkkotunnusta ja että validiusjakso on voimassa. Lopuksi selain käyttää saamansa julkista avainta varsinaisen istuntokohtaisen salausavaimen jakamiseen palvelimen kanssa. (Durumeric ym. 2013)

## 2.3 HTML

Web-sovelluksen tyypillinen sisältötyyppi on HTML, joka HTTP-vastauksessa ilmaistaan `Content-Type`-otsikkotietueessa termillä `text/html`. HTML-dokumentti koostuu elementtipuusta ja tekstistä (Hickson ym. 2014), jonka selain jäsentää ja näyttää käyttäjälle saadessaan sen palvelimelta HTTP-vastauksessa. Elementit voivat olla esimerkiksi tekstiä, kuvia tai komentojonoja.

Listauksessa 2.3 on esimerkki HTML-dokumentista, jossa esiintyvät elementit `html`, `head`, `title`, `body`, `h1` ja `p`. Elementti `head` sisältää dokumentin metatietoja, kuten tässä tapauksessa sivun otsikon. Elementti `body` puolestaan sisältää sivun näkyvän osan - tässä

tapauksessa otsikkoelementin `h1` ja tekstikappale-elementin `p`. Dokumentin alussa oleva `<!DOCTYPE html>` ilmoittaa, että dokumentti noudattaa HTML5-standardia.

```
<!DOCTYPE html>
<html>
 <head>
  <title>Esimerkkisivu</title>
 </head>
 <body>
  <h1>Esimerkkisivu</h1>
  <p>Tämä on esimerkkisivu.</p>
 </body>
</html>
```

Listaus 2.3. Esimerkki HTML-dokumentista.

## 2.4 Staattisista web-sivuista web-sovelluksiin

Internetin alkuaikoina web koostui lähinnä staattisista sivuista (Jazayeri 2007). Sen vuoksi niitä voikin sanoa ennemmin web-*sivuiksi* kuin web-sovelluksiksi. Staattisuus tarkoittaa sitä, että web-sivu näkyy jokaiselle käyttäjälle samanlaisena ajankohdasta ja käyttäjästä riippumatta.

Web-sovellukset ovat sen sijaan *dynaamisia* (Jazayeri 2007). Web-sovellus tyypillisesti koostuu HTML-dokumentista, selaimessa ajettavista komentojonoista sekä palvelimella ajettavasta ohjelmakoodista, joka tavallisesti on yhteydessä tietokantaan (Jazayeri 2007). Tämä tarkoittaa sitä, että web-palvelin generoi ohjelmallisesti selaimelle lähetettävän web-sivun HTTP-pyynnön sisällön ja web-palvelimen senhetkisen tilan perusteella. Kyseinen web-sivu voi sisältää JavaScript-koodia, mikä mahdollistaa entistä paremman interaktiivisuuden: ladatun sivun sisältöä voidaan muuttaa ilman, että palvelimen kanssa tarvitsee kommunikoida, ja sivu voi tarvittaessa lähettää palvelimelle asynkronisia HTTP-pyyntöjä, jotka päivittävät sivun osia ilman, että koko sivua tarvitsee ladata uudelleen.

HTTP-protokolla on *tilaton* (Fielding ja Reschke 2014a), mikä tarkoittaa sitä, että kukin HTTP-pyyntö on riippumaton edellisistä pyynnöistä. Monet web-sovellukset vaativat kuitenkin käyttäjältä kirjautumisen, minkä jälkeen web-sovellus tarjoaa käyttäjälle enemmän tietoja ja toimintoja. Kuinka web-sovellus voi siis HTTP-pyynnön saadessaan tietää, onko käyttäjä kirjautuneena vai ei? Ratkaisu ongelmaan ovat *evästeet* (*cookies*). Ne ovat selai-

men tietovarastoon talletettavia tietueita, joita web-palvelin lähettää HTTP-vastauksen otsikkoriveissä (Barth 2011a). Kun käyttäjä esimerkiksi lähettää kirjautumistietonsa (tavallisesti käyttäjätunnus-salasana-parin) web-palvelimelle, palvelin tarkastaa kirjautumistiedot ja lähettää HTTP-vastauksessa `Set-Cookie`-otsikkorivin, jolla on jokin uniikki arvo. Selain tallentaa tämän arvon tietovarastoonsa ja lähettää sen jokaisen seuraavan HTTP-pyynnön mukana `Cookie`-otsikkorivillä web-palvelimelle. Lähetetyn evästeen perusteella palvelin tunnistaa käyttäjän.

# 3 OWASP Top 10 -haavoittuvuudet web-sovelluksissa

Luvussa kuvataan web-sovellusten yleisimmät haavoittuvuudet OWASP Top 10 -listan mukaan.

## 3.1 Määritelmä

*Haavoittuvuus* on sellainen vika tai puute järjestelmän suunnittelussa, toteutuksessa tai hallinnoinnissa, jonka hyväksikäyttö vaarantaa järjestelmän tietoturvan (Shirey 2007).

Haavoittuvuuksien olemassaolo voi johtua muun muassa heikoista salasanoista, ohjelmistovirheistä, haittaohjelmista, koodi-injektiosta tai tarkastamattomasta käyttäjän syötteestä (Vacca 2013, 541). Haavoittuvuudesta riippuen hyökkääjä saattaa pahimmillaan päästä käsiksi tietyn tai kaikkien web-sovelluksen käyttäjien tietoihin, esiintyä toisena käyttäjänä, tuhota osan tai kaikki web-sovelluksen tiedoista tai asentaa web-sovellukseen haittaohjelmia.

OWASP (Open Web Application Security Project) on vuodesta 2003 lähtien kartoittanut web-sovellusten yleisimpiä haavoittuvuuksia. Vuoden 2013 kymmenen pahimman haavoittuvuuden lista perustuu useilta yrityksiltä saatuihin tietojoukkoihin, jotka kattavat yli 500 000 haavoittuvuutta tuhansissa sovelluksissa (Williams ja Wichers 2013). Kyseiset haavoittuvuudet ovat seuraavat (Williams ja Wichers 2013):

1. Injektio
2. Viallinen käyttäjän tunnistaminen ja istunnon hallinta (*Broken Authentication and Session Management*)
3. Cross-Site Scripting
4. Turvattomat suorat objektiviittaukset (*Insecure Direct Object References*)
5. Turvaton konfiguraatio (*Security Misconfiguration*)
6. Sensitiivisen tiedon paljastuminen (*Sensitive Data Exposure*)
7. Puuttuva ominaisuustason oikeushallinta (*Missing Function Level Access Control*)
8. Cross-Site Request Forgery
9. Haavoittuvien komponenttien käyttäminen (*Using Components with Known Vulnera-*

*bilities*)

10. Tarkastamattomat uudelleenohjaukset ja edelleenlähetykset (*Unvalidated Redirects and Forwards*)

Seuraavissa alaluvuissa kuvataan tarkemmin edellä mainitut haavoittuvuudet.

## 3.2 SQL-injektio

Tyypillinen web-sovellus käyttää jotakin tietokantaa hakeakseen ja tallentaakseen tietoa. Voidakseen kommunikoida tietokannan kanssa web-sovelluksen monesti täytyy rakentaa merkkijonomuotoinen komento, joka annetaan suoritettavaksi tietokannalle. *Injektio* syntyy, kun käyttäjän syöte päätyy osaksi kyseistä komentoa siten, että komennon rakenne muuttuu oleellisesti.

Tässä alaluvussa perehdytään esimerkin vuoksi *SQL-injektioon*, joka on yksi yleisimmistä injektiohaavoittuvuuksista. SQL (Structured Query Language) on tekstimuotoinen kieli, jota käytetään kommunikointiin relaatiotietokantojen kanssa. 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.

SQL-injektio tarkoittaa sitä, että hyökkääjä muuttaa WWW-sovelluksessa olevan SQL-kyselyn rakennetta lisäämällä kyselyyn omia SQL-avainsanoja tai operaattoreita (Halfond, Viegas ja Orso 2006). Listauksessa 3.1 on esimerkki SQL-injektiohaavoittuvuudesta pseudokoodissa. Haavoittuvuus johtuu siitä, että käyttäjän syöttämä `username`-parametri liitetään sellaisenaan osaksi SQL-kyselyä.

```
username = http_post_vars["username"]
sql = "SELECT name FROM users WHERE name LIKE '$username'"
result = db_execute(sql)
```

Listaus 3.1. Esimerkki SQL-injektiolle haavoittuvaisesta koodista.

Hyökkääjä voisi käyttää tätä haavoittuvuutta hyväkseen syöttämällä `username`-parametrin arvoksi esimerkiksi `' OR 1=1 --`. Tällöin suoritettava SQL-lause tulisi muotoon

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

jolloin kyseisen lauseen `WHERE`-ehdon lauseke saa aina arvon tosi, jolloin tulosjoukossa on kaikkien `users`-taulun käyttäjien nimet. Tilanteesta riippuu, miten paljon vahinkoa injektiolla voi saada aikaan. Hyökkääjä saattaisi esimerkiksi saada selville kaikkien `users`-taulussa olevien käyttäjien nimet, mikäli tulosjoukko lähetetään HTTP-vastauksessa selaimelle ilman erillisiä tarkistuksia.

### 3.2.1 Injektiomekanismit

On olemassa useita mekanismeja, joiden kautta SQL-injektio tapahtuu. Näistä yleisimpiä ovat käyttäjän syöte, evästeet sekä palvelinmuuttujat. (Halfond, Viegas ja Orso 2006) Listauksessa 3.1 mainittu tapaus on esimerkki käyttäjän syötteeseen perustuvasta injektiohaavoittuvuudesta. Useimmissa SQL-injektiohyökkäyksissä käyttäjän syötteet ovat peräisin WWW-sivulla olevasta lomake-elementistä (Halfond, Viegas ja Orso 2006).

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 (Halfond, Viegas ja Orso 2006).

Kolmas SQL-injektiomekanismi on palvelinmuuttujien hyödyntäminen. Palvelinmuuttujat ovat WWW-palvelimen itsensä generoimia, mutta ne voivat silti sisältää tietoja, jotka ovat lähtöisin käyttäjältä. Tällaisia ovat esimerkiksi HTTP-pyynnön parametriosa sekä HTTP-pyynnön osoite. Jos WWW-sovellus tallentaa palvelinmuuttujia sellaisenaan tietokantaan, se voi olla haavoittuvainen SQL-injektiolle (Halfond, Viegas ja Orso 2006).

SQL-injektiot voidaan luokitella karkeasti kahteen luokkaan sen mukaan, milloin vahinko tapahtuu. Ensimmäisen asteen injektiossa vahinko tapahtuu heti, kun hyökkääjän syöte saavuttaa tietokannan. Listauksessa 3.1 on esimerkki tällaisesta injektiosta. Toisen asteen SQL-injektiossa käyttäjän syöte tallennetaan aluksi oikein tietokantaan, mutta sitä käytetään toisessa SQL-kyselyssä myöhemmin, jolloin varsinainen injektio tapahtuu (Halfond, Viegas ja Orso 2006).

```
1 uid = get_current_user_id()
2 item = escape_sql(http_post_vars["item"])
3 db_execute("INSERT INTO items (uid, name) VALUES($uid, '$item')")
4 //Myöhemmin koodissa:
5 item_name = db_execute("SELECT name FROM items WHERE uid = $uid")
6 db_execute("SELECT uid FROM items WHERE name LIKE '$item_name'")
```

Listaus 3.2. Esimerkki toisen asteen SQL-injektiosta.

Listauksessa 3.2 on esimerkki toisen asteen SQL-injektiosta. Rivillä 2 käyttäjän syöttämä parametri `item` eskapoidaan (*escape*). Tämä tarkoittaa sitä, että syötteessä olevat SQL-erikoismerkit (kuten heittomerkki `'`) pakotetaan tulkittavaksi tavallisina merkkeinä, kun eskapoitu merkkijono liitetään osaksi SQL-lausetta. Eskapoinnin jälkeen syöte tallennetaan tietokantaan rivillä 3. Kyseinen tavara haetaan myöhemmin tietokannasta rivillä 5, ja se tallennetaan muuttujaan `item_name`. Injektiohaavoittuvuus syntyy siitä, että muuttujaa `item_name` ei ole eskapoitu, ja se liitetään sellaisenaan osaksi uutta SQL-kyselyä rivillä 6.

Jos esimerkiksi käyttäjä syöttää HTTP-pyynnön `item`-parametrissa merkkijonon `' OR 1=1 --`, niin rivin 2 suorituksen jälkeen muuttujaan `item` on tallennettu eskapoitu merkkijono `'' OR 1=1 --`. Tässä siis heittomerkki pakotetaan tavalliseksi merkiksi toistamalla se. Tällöin rivillä 3 tietokantaan tallentuu merkkijono `' OR 1=1 --`. Rivillä 5 kyseinen merkkijono haetaan tietokannasta, jolloin muuttujaan `item_name` tallentuu merkkijono `' OR 1=1 --`. Kyseisessä merkkijonossa on siis vain *yksi* heittomerkki, jolloin liitettäessä se osaksi SQL-kyselyä rivillä 6 se tulkitaan merkkijonon päättävänä merkkinä ja siten SQL-injektio tapahtuu.

### 3.2.2 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 (Halfond, Viegas ja Orso 2006). Yleisimmät tyypit ovat tautologiaan perustuva hyökkäys, UNION-hyökkäys ja reppuselkäkysely.

Tautologiaan perustuvassa hyökkäyksessä ehdolliseen SQL-lauseeseen liitetään sellainen osa, joka tekee ehtolausekkeesta aina toden (Halfond, Viegas ja Orso 2006). Listauksen 3.1 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 (Halfond, Viegas ja Orso 2006). 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 (Halfond, Viegas ja Orso 2006). Esimerkiksi listauksessa 3.1 hyökkääjä voisi syöttää `username`-parametrina merkkijonon `';DROP TABLE users--`, jolloin muuttujaan `sql` tallentuisi kaksi eri SQL-lausetta:

```
SELECT name FROM users WHERE name LIKE '';DROP TABLE users--'
```

Jälkimmäinen (eli puolipisteen jälkeinen) lause tuhoaisi tietokannassa olevan koko `users`-taulun sisällön. Kaikki tietokantatoteutukset eivät kuitenkaan tue useamman kuin yhden tietokantalauseen suorittamista yhdellä kertaa, joten tämän tyyppinen hyökkäys ei välttämättä toimisi.

### 3.2.3 SQL-injektioiden ehkäiseminen

SQL-injektiohaavoittuvuudet ovat seurausta siitä, etteivät sovelluskehittäjät ole noudattaneet turvallisia ohjelmointikäytänteitä (L. Shar ja Tan 2013). Injektioiden ehkäisemiseksi on ole-

massa kaksi tapaa: parametrisoidut kyselyt sekä syötteen eskapointi.

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

```
1 username = http_post_vars["username"]
2 stmt = prepare_sql("SELECT name FROM users WHERE name LIKE ?", username)
3 db_execute(stmt)
```

Listaus 3.3. Esimerkki parametrisoitujen kyselyjen käytöstä.

Listauksessa 3.3 on listauksen 3.1 kysely parametrisoidussa muodossa. Kyselyn muotti (*prepared statement*) on aliohjelman `prepare_sql` ensimmäinen parametri, jossa kysymysmerkki kertoo siihen sidottavan parametrin paikan. Siihen liitetään `username`-parametri.

Parametrisoidun SQL-kyselyn rakenne on ohjelmoijan ennalta määräämä, jolloin minkäänlainen SQL-injektio ei ole mahdollinen (L. Shar ja Tan 2013). Ohjelmoijan ei siis tarvitse itse huolehtia käyttäjän syöttämien parametrien eskapoinnista, vaan se tapahtuu automaattisesti.

Hyvin vanhoissa web-sovelluksissa ei kuitenkaan ole välttämättä mahdollista käyttää parametrisoituja kyselyitä, sillä siihen liittyviä ohjelmakirjastoja ei välttämättä ole saatavilla. Silloin ohjelmoijan on luotava SQL-kysely dynaamisesti liittäen merkkijonoja yhteen. Esimerkiksi listauksen 3.1 SQL-lause

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

on sama kuin

```
sql = "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 eskapoida käyttäjän syötteet (L. Shar ja Tan 2013). Tavallisesti ohjelmointikielen kirjasto tarjoaa tällaisen funktion valmiina. Ohjelmoijan on itse muistettava eskapoida kaikki käyttäjän syötteet käyttäen ohjelmointiympäristön tarjoamia eskapointimenetelmiä (L. Shar ja Tan 2013). Toisen

asteen injektioiden ehkäisemiseksi tämä koskee myös niitä tapauksia, joissa tietokannasta haettu käyttäjän syöttämä data liitetään osaksi toista SQL-kyselyä. Omatekoisten eskapointimenetelmien käyttäminen ei luonnollisesti ole suositeltavaa, sillä niihin voi jäädä ohjelmointivirheitä.

## 3.3 Käyttäjän tunnistamiseen liittyviä haavoittuvuuksia

Monet web-sovellukset vaativat käyttäjältä tunnistautumisen. Tavallisin käyttäjän tunnistamistapa on vaatia käyttäjältä tunnus ja salasana, jotka syötetään HTML-lomakkeelle (Stuttard ja Pinto 2011, 160). Käyttäjätunnuksiin ja salasanoihin liittyviä potentiaalisia haavoittuvuuksia ovat ainakin seuraavat (Stuttard ja Pinto 2011, 161):

- Huonojen salasanojen salliminen: Web-sovellus ei tarkasta käyttäjän määrittämän salasanan laatua: se voi olla hyvin lyhyt, tyhjä tai sama kuin käyttäjätunnus.
- Arvuuteltavissa oleva kirjautumistoiminto: Web-sovellus ei rajoita kirjautumisyritysten määrää tai tiheyttä, jolloin hyökkääjän on mahdollista arvuutella jopa tuhat salasanaa minuutissa.
- Liian yksityiskohtaiset virheviestit: Jos esimerkiksi käyttäjä syöttää salasanan väärin, web-sovellus ilmoittaa, että ”salasana on virheellinen”, mikä paljastaa hyökkääjälle, että käyttäjätunnus on olemassa.
- Puutteet salasanan vaihtotoiminnossa: Kyseinen toiminto puuttuu joko kokonaan tai siinä on haavoittuvuuksia (kuten nykyisen salasanan arvauskertojen rajoittamattomuus).
- Puutteet unohdetun salasanan toiminnossa: Salasanan vaihto tapahtuu vastaamalla käyttäjän asettamaan kysymykseen, jonka vastaus on yleensä hyökkääjän huomattavasti helpommin arvattavissa kuin varsinainen salasana.
- Puutteellinen salasanan tarkastus: Web-sovellus tarkastaa syötetystä salasanasta vain $n$ ensimmäistä merkkiä.
- Käyttäjätietojen turvaton tallentaminen: Web-sovellus esimerkiksi tallentaa salasanan tietokantaan sellaisenaan tai käyttää huonoa tiivistefunktiota (esim. MD5 tai SHA-1).
- Turvaton käyttäjätietojen jakelu: Web-sovellus esimerkiksi lähettää rekisteröitymisen yhteydessä käyttäjätunnuksen ja salasanan sähköpostitse, eikä vaadi salasanan vaihtoa

kirjauduttaessa sovellukseen ensimmäistä kertaa.

- Turvaton käyttäjätietojen lähetys: Web-sovellus esimerkiksi käyttää salaamatonta HTTP-protokollaa kirjautumisessa.

### 3.3.1 Salasanojen tietoturvasta

Salasanasta lasketun tiivisteen tallentaminen tietokantaan salasanan itsensä sijaan parantaa tietoturvaa, sillä mahdollisen tietomurron jälkeen hyökkääjä ei välittömästi saa tietoonsa käyttäjien salasanoja vaan ainoastaan niitä vastaavat tiivisteet. Jos tiivistefunktio on kuitenkin heikko (ts. tiivisteen laskeminen salasanan perusteella on nopeaa), hyökkääjä voi yrittää käyttää raakaa voimaa tiivisteiden murtamiseen. Tämä tapahtuu generoimalla tiivisteitä salasanaehdokkaista ja vertaamalla tiivisteitä tietokannasta saatuun.

Esimerkiksi SHA256-tiivistefunktio on altis raa'an voiman hyökkäyksille, sillä SHA256-tiivisteen laskeminen salasanasta on hyvin nopea operaatio, ja se nopeutuu edelleen tietokoneiden laskentatehon kasvaessa. Sen sijaan esimerkiksi *bcrypt* (Provos ja Mazières 1999) on nimenomaan salasanoja varten suunniteltu tiivistefunktio, joka sallii tiivisteen laskemisen vaativuuden säätämisen.

Käyttäjillä on taipumus valita salasanoja, jotka liittyvät heihin itseensä (Taneski, Heričko ja Brumen 2014). Tällainen salasana voi olla esimerkiksi sukulaisen tai lemmikkieläimen nimi tai syntymäpäivä. Kyseisenkaltaiset salasanat ovat monesti helposti arvattavissa. Web-sovellukset pyrkivät ehkäisemään ongelmaa määrittämällä salasananmuodostussääntöjä (kuten minimipituus tai tiettyjen merkkiryhmien pakollisuus) ja/tai kertomalla käyttäjälle arvion salasanan turvallisuudesta salasanamittarilla (*password meter*).

On todettu, että salasananmuodostussäännöt ja salasanamittarit auttavat käyttäjiä luomaan parempia salasanoja (Taneski, Heričko ja Brumen 2014; Ur ym. 2012; Komanduri ym. 2011). Ur ym. (2012) havaitsivat tutkimuksessaan, että tiukat tarkastimet auttoivat käyttäjiä luomaan vaikeammin murrettavia salasanoja, kun taas löysemmät tarkastimet eivät juuri parantaneet salasanojen laatua. Esimerkiksi Dropbox, Inc. -yhtiön kehittämän *zxcvbn*-salasanamittarin todettiin antavan järkeviä vasteita testaussanastoihin sen kehittyneen hahmontunnistuksen ansiosta (Carnavalet ja Mannan 2015).

## 3.4 Istunnon hallinnan haavoittuvuudet

Istunnon hallinta (*Session Management*) tarkoittaa niitä menetelmiä, joilla web-sovellus tunnistaa käyttäjän web-sovellukseen kirjautumisen jälkeen. Ilman istunnon hallintaa käyttäjä joutuisi syöttämään käyttäjätunnuksen ja salasanan jokaisen tunnistautumista vaativan sivun yhteydessä (Stuttard ja Pinto 2011, 206). Istunnon hallinta toteutetaan tavallisesti antamalla käyttäjälle uniikki istuntotunniste (*Session ID*), jota voidaan kuljettaa evästeissä, lomakkeiden näkymättömissä kentissä tai web-sovelluksen generoimissa linkeissä (Visaggio ja Blasio 2010).

Istunnon hallinnan haavoittuvuudet voivat liittyä joko istuntotunnisteiden generointiin tai kyseisten tunnisteiden käsittelyyn niiden elinaikana (Stuttard ja Pinto 2011, 207; Visaggio ja Blasio 2010). Jos hyökkääjän onnistuu kaapata tai arvata voimassa oleva istuntotunniste, hän saa tunnisteeseen liitetyn käyttäjän oikeudet web-sovelluksella itselleen kyseisen istunnon ajaksi.

### 3.4.1 Haavoittuvuudet istuntotunnisteen generoinnissa

Monet istunnon hallinnan haavoittuvuudet johtuvat siitä, että istuntotunnisteet generoidaan turvattomalla tavalla, jolloin hyökkääjän on mahdollista arvata tai päätellä niiden arvo. Stuttard ja Pinto (2011, 210) kuvaavat seuraavat kolme potentiaalisesti haavoittuvaa tapaa generoida istuntotunnisteita:

- Merkitykselliset tunnisteet (*Meaningful Tokens*): Tunniste on generoitu pelkästään käyttäjän tietojen perusteella. Jos tunniste esimerkiksi generoidaan käyttäjätunnuksen ja päivämäärän perusteella, hyökkääjän on mahdollista arvuutella suuri joukko potentiaalisesti valideja tunnisteita.
- Arvattavissa olevat tunnisteet (*Predictable Tokens*): Vaikka tunniste ei olisi merkityksellinen, siinä voi silti olla havaittavissa johdonmukaisuutta, jonka perusteella hyökkääjä voi päätellä tunnetun tunnistejoukon perusteella uusia tunnisteita. Pahimmassa tapauksessa tunniste on juokseva numero, jolloin validien tunnisteiden arvaaminen on triviaalia.
- Salatut tunnisteet (*Encrypted Tokens*): Tunnisteen sisältö (käyttäjän tiedot) salataan

huonolla salausalgoritmilla, jolloin hyökkääjän voi olla mahdollista muokata tunnisteen sisältöä.

### 3.4.2 Haavoittuvuudet istuntotunnisteen käsittelyssä

Vaikka tunniste generoitaisiin turvallisesti, web-sovelluksessa voi olla muita tunnisteen käsittelyyn liittyviä haavoittuvuuksia, joiden takia hyökkääjä saattaa saada sen selville (Stuttard ja Pinto 2011, 233; Visaggio ja Blasio 2010). Näistä kerrotaan seuraavissa kappaleissa.

Jos web-sovellus käyttää salaamatonta HTTP-yhteyttä, istuntotunniste on kaapattavissa käyttäjän ja web-palvelimen välisestä verkkoliikenteestä. Jos salattua yhteyttä käytetään kirjautumisen yhteydessä muttei sen jälkeen, tunniste on kaapattavissa kirjautumista seuraavilla pyynnöillä. Lisäksi web-sovellus saattaa asettaa käyttäjälle tunnisteen välittömästi, kun hän vierailee sivustolla ensimmäistä kertaa, ja kyseinen tunniste voi säilyä kirjautumisen yhteydessä. Siten myös kirjautumista edeltävä kommunikaatio tulisi salata. (Stuttard ja Pinto 2011, 234)

Istuntotunniste saattaa toisinaan tallentua erilaisiin lokitiedostoihin, joihin hyökkääjä saattaa päästä käsiksi, jos ne eivät ole kunnolla suojattuja. Jos tunniste lähetetään URL-osoitteen parametrina eikä HTTP-pyynnön otsikkotiedoissa tai sisältöosassa, niin tunnisteen tallentuminen lokitiedostoon on todennäköisempää. (Stuttard ja Pinto 2011, 237)

Kolmas haavoittuvuus istuntotunnisteiden käsittelyssä liittyy siihen, millä tavalla tunnisteet yhdistetään käyttäjiin. Esimerkiksi web-sovellus saattaa sallia usean tunnisteen yhtäaikaisen voimassaolon samalle käyttäjälle, vaikka siihen ei olisi hyvää syytä. Tällöin istunnon kaappaamisen havaitseminen on vaikeampaa. (Stuttard ja Pinto 2011, 240)

Jos web-sovellus generoi saman tunnisteen samalle käyttäjälle joka kirjautumiskerralla, kyseessä on ns. staattinen tunniste. Tällöin kyseessä ei ole edes kunnollinen istunto, koska web-sovellus tunnistaa käyttäjän tunnisteen perusteella riippumatta siitä, onko tämä juuri kirjautunut sisään vai ulos. Siten myös hyökkääjä voi yrittää kaapata kenen tahansa rekisteröityneen käyttäjän tunnisteen eikä ainoastaan sisäänkirjautuneiden. (Stuttard ja Pinto 2011, 240)

Istunnon oikeaoppinen päättäminen on tärkeää, sillä muuten mahdollisella hyökkääjällä on enemmän aikaa yrittää kaapata tunniste. Istunnon päättämiseen voi myös liittyä haavoittuvuuksia. Jos web-sovelluksessa ei ole uloskirjautumistoimintoa, käyttäjän on mahdotonta päättää istuntoaan. Toisinaan uloskirjautumistoiminto tyhjentää istuntotunnisteen ainoastaan selaimen muistista, eli jos käyttäjä lähettää saman tunnisteen web-palvelimelle, niin palvelin hyväksyy sen. (Stuttard ja Pinto 2011, 241-242)

Istuntotunnisteeseen liittyvän evästeen liiallinen laajuus voi myös aiheuttaa haavoittuvuuden. Esimerkiksi jos osoitteessa `app.example.com` oleva web-sovellus asettaa evästeen verkkotunnukseksi osoitteen `example.com`, niin tällöin istuntotunniste lähetetään myös muihin sovelluksiin, esimerkiksi `app2.example.com`. Tällöin muiden sovellusten haltijat voivat päästä käsiksi tunnisteeseen, jos käyttäjä vierailee muissa sovelluksissa. (Stuttard ja Pinto 2011, 245)

### 3.4.3 Istunnon hallinnan suojaaminen

Turvallinen istunnon hallinta edellyttää istuntotunnisteiden generointia turvallisesti sekä niiden turvallista hallinnointia niiden olemassaolon ajan (Stuttard ja Pinto 2011, 248).

Generoitujen tunnisteiden on oltava sellaisia, että ne eivät anna hyökkääjälle mahdollisuutta päätellä käyttäjien tunnisteiden arvoja aiemmin havaittujen tunnisteiden perusteella. Tehokas generointialgoritmi on sellainen, joka generoi tunnisteen hyvin suuresta joukosta mahdollisia arvoja, ja sillä on hyvä satunnaisuuden lähde, jotta tunnisteiden arvot jakautuvat joukkoon tasaisesti ja ennalta arvaamattomasti. Tunnisteen ei siten pidä sisältää minkäänlaista informaatiota käyttäjän tietoihin liittyen. (Stuttard ja Pinto 2011, 248)

Hyvän satunnaisalgoritmin lisäksi tunnisteen generoinnissa tulisi hyödyntää sen HTTP-pyynnön tietoja, jota varten tunniste generoidaan. Näitä ovat esimerkiksi pyynnön IP-osoite, `User-Agent`-otsikkotietue sekä pyynnön aikaleima. Tämä ehkäisee mahdollisia satunnaisalgoritmin heikkouksia. Hyvä algoritmi on esimerkiksi sellainen, joka muodostaa satunnaisluvun, HTTP-pyynnön tietojen sekä salaisen merkkijonon yhdisteen, josta lasketaan SHA-256-tiiviste. Tämä tiiviste on vakiomittainen ja kelpaa tunnisteeksi. (Stuttard ja Pinto 2011, 249)

Tunnisteen generoinnin lisäksi sitä täytyy käsitellä turvallisesti sen olemassaolon aikana. Tunnistetta tulee kuljettaa vain salatun HTTPS-yhteyden yli, jotta verkkoliikenteestä ei ole mahdollista kaapata sitä. Silloinkaan sitä ei tule liittää URL-osoitteen osaksi, koska silloin se tallentuu suuremmalla todennäköisyydellä johonkin lokitiedostoon. Istunnon päättäminen tulee toteuttaa tarjoamalla käyttäjälle uloskirjautumistoiminto ja asettamalla istunnolle enimmäiskesto, jonka jälkeen istunto vanhenee, mikäli käyttäjä ei tee sivustolla mitään tämän ajan kuluessa. Käyttäjän kirjautuessa sovellukseen mahdolliset olemassa olevat istunnot tulee poistaa, eli käyttäjällä tulee olla vain yksi voimassa oleva istunto. Lopuksi istuntoon liittyvän evästeen laajuus tulisi olla mahdollisimman pieni. (Stuttard ja Pinto 2011, 250)

Myös istunnon hallintaan kohdistuvilta hyökkäyksiltä on suojauduttava. Esimerkiksi mielivaltaisia käyttäjän lähettämiä tunnisteita ei pidä hyväksyä, vaan tunnisteen oikeellisuus ja voimassaolo täytyy aina varmistaa. Kirjautumisen jälkeen tulisi aina luoda uusi istunto, jotta vältytään istunnon kiinnityshyökkäyksiltä. Lisäksi XSS-haavoittuvuuksia tulee välttää, sillä ne potentiaalisesti mahdollistavat istunnon kaappaamisen. (Stuttard ja Pinto 2011, 251)

## 3.5 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 tarkastamisesta. (L. K. Shar ja Tan 2012)

```
print("<div>Hello, " + http_get_vars["name"] + "!</div>")
```

Listaus 3.4. Yksinkertainen esimerkki XSS-haavoittuvuudesta.

Listauksessa 3.4 on esimerkki pseudokoodista, 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 (Barth 2011b). Kyseinen käytäntö rajoittaa lähdettä A lukemasta dataa eri lähteestä B tiettyjä poikkeuksia lukuun ottamatta (Barth 2011b).

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 (Kerschbaum 2007). WWW-palvelimet käyttävät evästeitä säilyttämään käyttäjän tilan sivustolla (Barth 2011a).

XSS-injektiossa voidaan käyttää hyväksi saman lähteen käytäntöä (Kerschbaum 2007), koska hyökkääjän injektoima koodi on samalla sivulla. Hyökkääjä voi esimerkiksi injektoida WWW-sivulle sinne kuulumatonta JavaScript-koodia käyttäen `script`-elementtiä. Näin on mahdollista varastaa käyttäjän evästeet, vakoilla käyttäjää tai kaapata käyttäjätili (Kerschbaum 2007; L. K. Shar ja Tan 2012).

### 3.5.1 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*) (L. K. Shar ja Tan 2012).

Heijastetussa XSS-hyökkäyksessä WWW-palvelin sisällyttää HTTP-pyynnössä käyttäjän antamaa syötettä WWW-sivuun, joka lähetetään HTTP-vastauksessa selaimelle (L. K. Shar ja Tan 2012). Listauksessa 3.4 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 merkkijono:

```
"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 (L. K. Shar ja Tan 2012). 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 (L. K. Shar ja Tan 2012). Esimerkiksi seuraava WWW-sivulla oleva komentosarja on altis hyökkäykselle:

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

Jos hyökkääjä saa käyttäjän napsauttamaan linkkiä, jonka `href`-attribuuttina on merkkijono

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

niin silloin selain suorittaa linkkitekstin `script`-elementissä olevan komentosarjan, mikäli selain ei automaattisesti eskapoi `document.write`-metodin merkkijonoparametria.

### 3.5.2 XSS-injektioiden ehkäiseminen

XSS-injektiot voidaan torjua riittävällä syötteen tarkastamisella. Tähän on olemassa neljä perustapaa. Syötteessä olevat erikoismerkit voidaan korvata toisilla merkeillä tai poistaa kokonaan. Erikoismerkit voidaan myös eskapoida. Neljäs keino on määritellä vain sallitut merkit, joita käyttäjä voi WWW-sovellukseen syöttää. (L. K. Shar ja Tan 2012)

Ongelma erikoismerkkien muokkaamisessa tai poistamisessa on se, että käyttäjän WWW-sovellukseen lähettämän viestin alkuperäinen merkitys voi hämärtyä. Sallittujen merkkien rajoittaminen voi puolestaan johtaa monen käyttäjän syötteen hylkäämiseen (L. K. Shar ja Tan 2012). Nämä ongelmat voidaan välttää syötteen eskapoinnilla.

Yleisessä tapauksessa käyttäjän syötteen eskapointi XSS-injektion ehkäisemiseksi ei ole yksinkertaista. Syötteen oikeanlainen eskapointi tarvitsee tiedon siitä, missä rakenteellisessa kohdassa käyttäjän syöte esiintyy WWW-sivulla (Weinberger ym. 2011). Esimerkiksi listauksessa 3.4 syöte esiintyy `div`-elementin sisällä. Silloin riittää eskapoida HTML-erikoismerkit (Weinberger ym. 2011). Toisaalta käyttäjän syöte voisi olla `href`-attribuutin sisällä:

```
print('<a href="' + http_post_vars["address"] + '">Test</a>')
```

Tällöin HTML-erikoismerkkien eskapointi ei ole enää riittävä suojauskeino XSS-injektiota vastaan, sillä se ei riitä suojaamaan muun muassa `javascript:`-alkuisia URI-osoitteita vastaan (Weinberger ym. 2011). Tämä pätee vain siinä tapauksessa, että käyttäjän syöte muodostaa koko linkin. Silloin ennen syötteen eskapoimista on tarkastettava, että syötetyn linkin protokolla on kelvollinen (OWASP 2012), 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 (Weinberger ym. 2011). Listauksessa 3.5 on esimerkki tällaisesta tilanteesta, jossa syöte on toisaalta heittomerkkien sisällä JavaScript-koodissa, mutta myös `script`-elementin sisällä.

```
print("<script>var page = '" + http_post_vars["page"] + "'</script>")
```

Listaus 3.5. Esimerkki käyttäjän syötteestä sisäkkäisessä rakenteessa.

Tässä tilanteessa hyökkääjä voisi käyttää syötteessään joko heittomerkkiä tai lopettavaa `</script>`-tagia aiheuttaakseen injektion (Weinberger ym. 2011).

On olemassa myös useita muita rakenteita, joissa käyttäjän syöte tarvitsee tietynlaisen eskapoinnin, mutta useimmissa tapauksissa riittää noudattaa seuraavaa kahta sääntöä (OWASP 2012):

- 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 eskapoitava.

## 3.6 Turvattomat suorat objektiviittaukset

Web-sovelluksella on tavallisesti objekteja, jotka ovat kenen tahansa ladattavissa: kuvat, JavaScript-tiedostot ja CSS-tiedostot. Näihin viitataan tavallisesti suoralla objektiviittauk-

sella, mikä tarkoittaa sitä, että objektin URL-osoitteessa oleva tiedostopolku vastaa web-palvelimen tiedostojärjestelmässä olevaa hakemistorakennetta. Tällainen viittaus on turvaton, jos objekti on kenen tahansa ladattavissa ja jos se sisältää sellaista tietoa, johon vain tiettyjen käyttäjien kuuluisi päästä käsiksi.

Jos esimerkiksi web-sovellukseen tallennetaan dokumentteja, ja käyttäjän oman dokumentin URL-osoite on muotoa

```
http://example.com/pdf/document1.pdf
```

niin käyttäjä voisi tällöin yrittää arvata muiden käyttäjien dokumenttien osoitteet.

Turvattomien suorien objektiviittausten ehkäisemiseksi on kaksi tapaa (Williams ja Wichers 2013):

- Käyttäjä- tai istuntokohtaisten objektiviitteiden käyttäminen: Sen sijaan, että web-sovellus paljastaa suorat objektiviitteet, voidaan käyttäjän istuntoon tallentaa istuntokohtaiset viitteet, jolloin käyttäjä ei koskaan näe suoria viitteitä.
- Pääsyoikeuden tarkastaminen: Aina, kun käyttäjä yrittää saada pääsyn johonkin objektiin suoralla viitteellä, pääsyoikeus on tarkastettava.

## 3.7 Turvaton konfiguraatio

Web-sovelluksen asettaminen toimintakuntoon vaatii monesti useiden eri asetuksien määrittämistä. Näitä voivat esimerkiksi olla erilaiset käyttäjätunnukset ja salasanat, virheviestien näkyvyys sekä mahdolliset sovelluskehyskohtaiset asetukset. Huonot asetukset voivat tehdä sovelluksesta haavoittuvan. Esimerkiksi liian yksityiskohtaiset virheviestit voivat paljastaa hyökkääjälle tietoja sovelluksen tilasta, ja oletussalasanat ovat helposti arvattavissa.

Konfiguraatiovirheiden välttämiseksi sovelluksen käyttöönotto tulisi olla mahdollisimman automatisoitua, ja kehitys- ja tuotantoympäristön tulisi olla identtisesti konfiguroituja, pois lukien eri salasanat. Lisäksi sovellusarkkitehtuurissa eri komponentit tulisi selvästi erottaa toisistaan, ja turvatarkastuksia tulisi suorittaa sovellukselle säännöllisesti. (Williams ja Wichers 2013)

## 3.8 Sensitiivisen tiedon paljastuminen

On vaikeaa määritellä, mitä sensitiivinen tieto on (Al-Fedaghi 2012). Magnabosco (2009, 20–21) määrittelee sen tiedoksi, joka yksikäsitteisesti identifioi henkilön tai jonka voidaan katsoa olevan henkilökohtaista. Esimerkiksi henkilötunnus ja potilastiedot ovat tällaisia. Sensitiivisen tiedon piiriin kuuluu myös sellainen tieto, joka paljastuessaan voisi aiheuttaa vahinkoa yksilölle, yritykselle tai yhteiskunnalle (Magnabosco 2009, 21). Näitä voivat olla esimerkiksi käyttäjätilien salasanat ja yrityksen tietojärjestelmän lähdekoodi.

Sensitiivinen tieto vaatii suojauksen, jotta sen paljastumista voidaan ehkäistä (Magnabosco 2009, 21). Toisaalta Al-Fedaghi (2012) määrittelee tiedon sensitiivisyyden määräytyvän sen vaatimasta suojaustasosta, joka puolestaan määräytyy niiden riskien perusteella, jotka voisivat toteutua tiedon väärinkäytöstä tai paljastumisesta johtuen.

Sensitiivisen tiedon suojaamiseksi tulisi suorittaa vähintään seuraavat asiat (Williams ja Wichers 2013):

- Sensitiivinen tieto tulee salata sekä varastoituna että silloin kun sitä siirretään.
- Sensitiivistä tietoa ei tule varastoida, ellei se ole välttämätöntä.
- Salaukseen tulee käyttää standardeja algoritmeja sekä vahvoja salausavaimia. Lisäksi avaimenhallinnan on oltava asianmukaista.
- Salasanat tulee varastoida käyttäen jotain salasanojen suojausta varten suunniteltua algoritmia.
- Sensitiivistä tietoa kerääviltä lomakkeilta on otettava automaattitäydennysominaisuus pois käytöstä. Lisäksi välimuisti tulisi ottaa pois käytöstä sivuilta, jotka sisältävät sensitiivistä tietoa.

## 3.9 Puuttuva ominaisuustason oikeushallinta

Web-sovelluksissa on tavallisesti useita funktioita, joista osa on näkyvillä vain niille käyttäjille, joilla on oikeus käyttää kyseisiä funktioita. Sovelluksessa voi kuitenkin olla haavoittuvuus, joka sallii funktion käyttämisen lähettämällä funktioon liittyvän HTTP-pyynnön palvelimelle. Esimerkiksi dokumentinhallintasovelluksessa käyttäjillä voi olla erikseen dokumen-

tin luku- ja muokkausoikeudet, ja käyttäjät, joilla ei ole muokkausoikeutta, eivät näe sivulla muokkausfunktiota. Jos oikeudeton käyttäjä kuitenkin lähettää HTTP-pyynnön muokkausfunktioon, niin haavoittuvuudesta johtuen muokkauspyyntö hyväksytään. Muokkausfunktiosta on siis unohtunut käyttäjän oikeuksien tarkastus.

Eräs mahdollinen tapa ehkäistä oikeuksien tarkastamiseen liittyviä haavoittuvuuksia on olettaa, että jokainen web-sovelluksen funktio vaatii korkeimmat oikeudet toimiakseen. Jos funktiota varten tarvitsee vähemmät oikeudet, niin tämä ilmaistaan eksplisiittisesti. Tällöin oikeuksien tarkastamista ei tarvitse erikseen muistaa, vaan ohjelmoijan täytyy ainoastaan lieventää oikeusvaatimuksia tarvittaessa.

## 3.10 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF)-hyökkäyksessä käyttäjän selain saadaan lähettämään käyttäjän tietämättä HTTP-pyyntö toiselle sivustolle (Zeller ja Felten 2008). CSRF voi tapahtua käyttäjän vieraillessa haitallisella sivustolla, joka käskee selainta tekemään pyynnön toiselle sivustolle (Barth, Jackson ja Mitchell 2008). Hyökkäyksen onnistuminen johtuu siitä, ettei pyyntöä vastaanottava sivusto tarkasta, mistä lähteestä pyyntö on peräisin. Sivusto siis luulee, että pyyntö lähetettiin käyttäjän omasta toimesta (Zeller ja Felten 2008).

Monet WWW-sivustot ovat haavoittuvia CSRF-hyökkäyksille, vaikka niiden torjuminen on helppoa. Haavoittuvuudet johtuvat siitä, että sovelluskehittäjät ovat tietämättömiä CSRF-haavoittuvuuden syistä ja seurauksista. Lisäksi saman lähteen käytäntö ei anna suojaa CSRF-hyökkäyksiä vastaan, koska se ei kiellä eri lähteitä lähettämästä pyyntöjä toisilleen (Zeller ja Felten 2008).

### 3.10.1 Esimerkki CSRF-haavoittuvuudesta

Oletetaan, että käyttäjän tuntemalla sivustolla on listauksessa 3.6 oleva lomake.

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

Listaus 3.6. Esimerkki WWW-sivulla olevasta lomakkeesta.

Käyttäjän napsauttaessa `Send message`-painiketta selain lähettää HTTP-pyynnön osoitteeseen `http://site.com/sendmsg?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`-elementin `src`-attribuutissa. Silloin käyttäjän riittäisi vierailla hyökkäyssivustolla, jolloin selain lähettäisi pyynnön sivustolle yrittäessään ladata kuvaa (Zeller ja Felten 2008).

CSRF-haavoittuvuus on sitä vakavampi, mitä enemmän käyttäjällä on oikeuksia sivustolla (Zeller ja Felten 2008). Tämä johtuu siitä, että HTTP-pyynnössä sivustolle lähetetään sivustoon liittyvät evästeet, vaikka pyyntö olisi lähtöisin hyökkäyssivustolta. Hyökkääjä pystyy siis tekemään toimintoja sivustolla aivan kuin sisäänkirjautunut käyttäjä.

### 3.10.2 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 (Zeller ja Felten 2008). Tämä sääntö noudattaa myös HTTP 1.1-protokollaa, jonka mukaan GET-metodin tulisi ainoastaan noutaa tietoa palvelimelta (Fielding ym. 1999). 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 (2008) ehdottavat, että jokaisen POST-pyynnön mukana tulee lähettää satunnaisluku, joka on vaikeasti arvattavissa. Tämän satunnaisluvun täytyy olla sama kuin sivuston evästeessä oleva. Koska hyök-

kääjä ei voi lukea käyttäjän evästeitä saman lähteen käytännöstä johtuen, CSRF-hyökkäys toteutuu vain, jos hyökkääjä onnistuu arvaamaan kyseisen satunnaisluvun. (Zeller ja Felten 2008)

Generoidun satunnaisluvun arvo ei saa riippua käyttäjätilistä, jotta se torjuisi myös ns. sisäänkirjautumis-CSRF-hyökkäyksen (*Login CSRF*) (Zeller ja Felten 2008). Tällaisessa hyökkäyksessä hyökkääjä saa käyttäjän kirjautumaan sisään WWW-palveluun hyökkääjän omilla tunnuksilla. Tällä tavalla hyökkääjä voi esimerkiksi vakoilla käyttäjän toimia kyseisellä sivustolla. (Barth, Jackson ja Mitchell 2008)

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

Vaihtoehtoinen tapa CSRF-hyökkäyksen torjumiseksi on tarkastaa HTTP-pyynnön Referer-otsikkotietue (Barth, Jackson ja Mitchell 2008). Tämä otsikkotietue kertoo, miltä WWW-sivulta pyyntö on peräisin (Fielding ym. 1999). Kyseistä tietuetta ei voi väärentää JavaScript-koodin avulla. Poikkeuksen muodostavat jotkin vanhat selaimet, joiden sisältämien haavoittuvuuksien vuoksi Referer-tietueen väärentäminen on mahdollista (Barth, Jackson ja Mitchell 2008). CSRF-hyökkäyksen torjumiseksi riittää siis tarkastaa, että pyyntö on lähtöisin samalta sivustolta kuin johon pyyntö kohdistuu (Barth, Jackson ja Mitchell 2008).

Referer-otsikkotietueen käyttäminen CSRF-hyökkäyksen torjumiseksi ei kuitenkaan ole täysin ongelmatonta. Kyseinen otsikkotietue voi toisinaan vuotaa käyttäjästä tietoa (Krishnamurthy ja Wills 2009). Siksi jotkut internetin käyttäjät ovat estäneet sen lähetyksen HTTP-pyynnön mukana (Barth, Jackson ja Mitchell 2008). Otsikkotietueen puuttuessa sivusto ei voi enää päätellä pyynnön alkuperää, joten se joutuu näyttämään virheilmoituksen.

## 3.11 Haavoittuvien komponenttien käyttäminen

Web-sovellus käyttää tavallisesti monia eri ohjelmakirjastoja toimiakseen. Jos jokin kirjasto sisältää haavoittuvuuksia, myös web-sovellus voi olla haavoittuvainen, vaikka kirjastoa

käytettäisiin sen ohjeiden mukaisesti. Siksi haavoittuvan komponentin käyttöä tulee välttää.

Ei ole aina helppoa saada selville, käyttääkö web-sovellus mitään haavoittuvia komponentteja, koska eri komponenttien haavoittuvuuksia ei aina raportoida keskitetysti. On kuitenkin olemassa toimenpiteitä, jotka auttavat tämän ehkäisemisessä. Kehittäjien tulisi esimerkiksi ylläpitää listaa kaikista web-sovelluksen käyttämistä komponenteista sekä myös näiden riippuvuuksista. Komponentteihin liittyviä turvallisuusilmoituksia tulisi seurata julkisten tietokantojen ja sähköpostilistojen avulla, mikäli mahdollista. Komponentin ympärille voi lisäksi muodostaa kääreen, jonka avulla komponentin toimintaa voi tarvittaessa rajoittaa. (Williams ja Wichers 2013)

## 3.12 Tarkastamattomat uudelleenohjaukset ja edelleenlähetykset

Uudelleenohjaus (*redirect*) on tapa ilmoittaa, että selaimen tulisi siirtyä toiselle sivulle. Tavanomainen käyttötapaus on esimerkiksi se, että web-sovellus ohjaa käyttäjän kirjautumisen jälkeen sille sivulle, jossa hän oli ennen kirjautumissivulle tuloa. Tämän toteuttamiseksi kirjautumissivulle tultaessa selaimelle on jollain tavalla lähetettävä paluusivun URL-osoite, joka kirjautumisen yhteydessä ilmoitetaan palvelimelle. Jos käyttäjä esimerkiksi päätyy kirjautumiseen sivulta `/profile`, niin paluusivun osoite voidaan liittää kirjautumissivun URL-osoitteen parametriksi:

```
https://example.com/login?returnTo=/profile
```

Uudelleenohjausmekanismi on haavoittuvainen, jos paluusivun osoitetta ei tarkasteta. Tällöin hyökkääjä voi esimerkiksi ohjata käyttäjän haitalliselle sivustolle muokkaamalla osoitteen määräävän parametrin haluamakseen.

Uudelleenohjaukseen on olemassa useita mekanismeja. Palvelin voi lähettää 3xx-tilakoodin, jolloin `Location`-otsikkotietue ilmoittaa ohjauksen kohdeosoitteen. Tavanomaisen tilakoodin 200 yhteydessä voi käyttää `Refresh`-otsikkotietuetta, jonka arvona on kohdeosoite sekä aikaraja, jonka kuluttua ohjaus suoritetaan. Kolmas tapa on lisätä HTML-dokumentin `head`-elementtiin `meta`-elementti, joka toimii vastaavalla tavalla kuin `Refresh`-otsikkotietue. Ohjauksen voi toteuttaa myös käyttäen JavaScript-kieltä muuttamalla `document.location`-

attribuuttia. (Stuttard ja Pinto 2011, 542)

Web-sovellus voi käyttää edelleenlähetystä (*forward*) reitittääkseen pyyntöjä sivuston eri osien välillä (Williams ja Wichers 2013). Edelleenlähetystä voidaan käyttää silloin, kun jonkin toiminnon suorittamisen jälkeen halutaan siirtyä tietylle sivulle. Tällöin toimintoon liittyvä URL-osoite voi olla esimerkiksi

```
https://example.com/action?forward=/home
```

jolloin toiminnon suorittamisen jälkeen siirryttäisiin `/home`-sivulle. Haavoittuvuus syntyy, jos web-sovellus ei tarkasta, onko käyttäjällä oikeus `forward`-parametrin kuvaamalle sivulle.

Haavoittuvuuksien välttämiseksi uudelleenohjauksissa ja edelleenlähetyksissä ei tulisi käyttää käyttäjän antamia parametreja. Mikäli tätä ei voi välttää, on varmistettava, että parametrien arvot ovat valideja ja että käyttäjällä on pääsy parametrissa mainittuun sivuun. (Williams ja Wichers 2013).

# 4 Haavoittuvuuksien etsiminen web-sovelluksista

Luvussa kuvataan tekniikat, joiden avulla web-sovelluksista voidaan tunnistaa haavoittuvuuksia. Näitä ovat white-box-, black-box- sekä gray-box -analyysit.

## 4.1 White-box-analyysi

*White-box-analyysi* tarkoittaa web-sovelluksen tarkastelemista sisältäpäin (Antunes ja Vieira 2014). Tällöin web-sovelluksen koodia ei tarvitse suorittaa. Analyysissa voidaan hyödyntää sovelluksen lähdekoodia, dokumentaatiota tai muuta materiaalia (Stuttard ja Pinto 2011, 702). White-box-analyysi voidaan toteuttaa automaattisesti tai manuaalisesti. Manuaalinen analyysi voi olla esimerkiksi lähdekoodin katselmointia, läpikäyntiä tai siihen voi liittyä formaalimpi turvatarkastus, jossa koodia vertaisarvioidaan systemaattisesti haavoittuvuuksien varalta (Antunes ja Vieira 2014).

Web-sovelluksen lähdekoodia on usein mahdollista analysoida myös automaattisilla työkaluilla. Kyseiseen menetelmään liittyy kuitenkin ongelmia. Kattava lähdekoodin analysoiminen voi olla vaikeaa, ja lisäksi automaattisilta työkaluilta jää usein huomaamatta monet haavoittuvuudet lähdekoodin monimutkaisuuden vuoksi (Antunes ja Vieira 2014).

Stuttard ja Pinto (2011, 703) ehdottavat seuraavaa kolmea toimenpidettä koodikatselmoinnin suorittamiseen:

- Käyttäjän syöttämän datan jäljittäminen ja sitä prosessoivan lähdekoodin katselmointi.
- Sellaisten lähdekoodin kohtien katselmointi, jotka potentiaalisesti indikoivat haavoittuvuuden olemassaoloa.
- Kriittisen lähdekoodin katselmointi rivi riviltä. Näitä ovat esimerkiksi istunnon hallinta, kulunvalvonta (*access control*) sekä käyttäjän syötteen validointi.

Potentiaalisesti haavoittuvia ohjelmakoodin kohtia ovat esimerkiksi ne kohdat, joissa tietokanta-ajurille syötettäviä SQL-lauseita tai HTML-merkintäkieltä muodostetaan merkkijonoja yhteen liittämällä.

## 4.2 Black-box-analyysi

*Black-box-analyysissä* eli penetraatiotestauksessa web-sovellusta tarkastellaan ulkoapäin siten, että mitkään sen sisäiset yksityiskohdat eivät ole analysoijan tiedossa (Antunes ja Vieira 2014). Web-sovellus on siis kuin musta laatikko, joka vastaa tiettyyn syötteeseen tietyllä tavalla. Haavoittuvuudet pyritään havaitsemaan analysoimalla web-sovelluksen antamia vasteita. Koska jokaista haavoittuvuustyyppiä varten voi joutua ajamaan jopa tuhansia eri syötteitä, manuaalinen testaaminen on hidasta (Antunes ja Vieira 2014). Siksi onkin kehitetty työkaluja, jotka suorittavat black-box-testaamista automaattisesti (Antunes ja Vieira 2014; Austin, Holmgreen ja Williams 2013).

Austin, Holmgreen ja Williams (2013) jakavat manuaalisen penetraatiotestauksen kahteen osaan: järjestelmälliseen ja eksploratiiviseen penetraatiotestaukseen. Ensimmäinen noudattaa ennalta määrättyä testaussuunnitelmaa, kun taas jälkimmäisessä sellaista ei ole, vaan testaaminen tapahtuu opportunistisesti ja se perustuu testaajan vaistoihin ja kokemuksiin (Austin, Holmgreen ja Williams 2013).

*Gray-box-analyysissä* yhdistetään white-box- ja black-box-tekniikoita. Tämä voi tarkoittaa esimerkiksi sovelluksen sisäisen tilan analysointia samalla kun sille annetaan realistisia syötteitä (Antunes ja Vieira 2014). Automaattiselle penetraatiotestaustyökalulle voidaan puolestaan antaa kuvaus web-sovelluksen käytössä olevista reiteistä ja niiden mahdollisista parametreista, jolloin työkalun ei tarvitse päätellä reittejä tai niiden parametreja itse.

# 5 TIM-järjestelmä

Luvussa esitellään TIM-järjestelmän maaliskuun 2015 versio (päiväys 13. maaliskuuta 2015), sen taustaa ja sen käytössä olevat tekniikat.

## 5.1 Taustaa

Monien kurssien web-opetusmateriaalissa on parantamisen varaa. Esimerkiksi opiskelijan on usein mahdotonta lisätä sähköiseen luentomonisteeseen omia muistiinpanojaan. Jos materiaalista korjataan virheellinen kohta, jonka opiskelija ehti jo lukea, ei hän tätä korjausta välttämättä jälkeenpäin havaitse. Mikäli materiaali on jaettu useammalle sivulle, sivuilla olevista linkeistä voi muodostua sekava kokonaisuus. Luentomateriaaliin liittyvät tehtävät ovat usein erillään, jolloin voi olla hankalaa havaita, mitkä tehtävät vastaavat mitäkin luentomateriaalin lukua. Lisäksi tehtävistä ja materiaalista puuttuu interaktiivisuus: opiskelijan on hankala varmistaa, onko hän ymmärtänyt lukemaansa, ja asioiden havainnollistaminen jää usein pelkän tekstin ja kuvien varaan.

Jyväskylän yliopiston tietotekniikan laitoksella on meneillään *The Interactive Material* -projekti, jonka tavoitteena on kehittää ohjelmisto ja malli e-kurssimateriaalien hallintaan ja kehitykseen. Kehitettävällä järjestelmällä voidaan ratkaista edellä mainittuja perinteisiin web-opetusmateriaaleihin liittyviä ongelmia. Järjestelmän avulla kurssimateriaali voidaan kirjoittaa lineaariseen muotoon, jolloin linkkikaaosta ei pääse syntymään. Tämän mahdollistaa mm. dokumenttiin upotettavat interaktiiviset komponentit, jotka voivat olla esimerkiksi ajettavaa ohjelmakoodia tai harjoitustehtäviä. Opiskelijat voivat lisäksi kommentoida ja/tai muokata materiaalia asetetuista oikeuksista riippuen. Järjestelmän tavoitteena on myös lisätä opettajan ja opiskelijoiden välistä vuorovaikutusta esimerkiksi erilaisten luentokysymysten avulla.

## 5.2 Tekniikat

TIM-sovellukseen liittyviä olennaisia komponentteja ja tekniikoita ovat Nginx, Docker, Flask ja AngularJS. Nämä kuvataan lyhyesti seuraavissa alaluvuissa.

### 5.2.1 Nginx

*Nginx* on suorituskykyinen web-palvelin, jonka toimintaa ohjataan konfiguraatiotiedostoilla. Kyseiset tiedostot sisältävät alueita (*section*), jotka sisältävät toisia alueita ja/tai direktiivejä (*directive*). Direktiivit, jotka eivät kuulu mihinkään alueeseen, ovat globaaleja. (Aivaliotis 2013, 1, 21–22)

```
user www;
error_log /var/log/nginx/error.log;
events {}
http {
    default_type application/octet-stream;
    server {
        listen 80;
        server_name www.example.com;
        location /static/ {
            alias /var/www/static/;
        }
        location / {
            proxy_pass http://127.0.0.1:8080;
        }
    }
}
```

Listaus 5.1. Esimerkki Nginxin konfiguraatiotiedostosta.

Listauksessa 5.1 on esimerkki Nginxin konfiguraatiotiedostosta. Globaaleja direktiivejä on kaksi: `user` määrittää käyttäjän, jonka alla palvelinprosessia suoritetaan, ja `error_log` määrittää virhelokin sijainnin tiedostojärjestelmässä. Pakollisessa alueessa `events` voidaan tarvittaessa määrittää yhteyden prosessointiin liittyviä direktiivejä (Aivaliotis 2013, 223). Alueessa `http` määritellyt direktiivit vaikuttavat HTTP-yhteyksiin. Direktiivi `default_type` ilmoittaa oletustyypin HTTP-vastauksille, jos Nginx ei pysty päättelemään sitä tiedostopäätteestä. Seuraavaksi määritellään virtuaalipalvelin, joka kuuntelee porttia 80. Direktiivi `server_name` vaatii, että HTTP-pyynnön `Host`-otsikkotietueen tulee olla `www.example.com`, jotta pyyntö ohjataan tähän virtuaalipalvelimeen.

Direktiivin `location` avulla voidaan määrätä asetuksia halutuille URL-osoitteille. Esimerkissä 5.1 kyseinen direktiivi esiintyy kaksi kertaa. Ensimmäinen koskee niitä URL-osoitteita, joiden polkuosa alkaa `/static/`. Direktiivin sisällä oleva `alias`-direktiivi määrää, että tällöin URL-osoitteen loppuosassa oleva tiedosto haetaan palvelimen `/var/www/static/`-hakemistosta. Jälkimmäinen `location`-direktiivi koskee kaikkia muita osoitteita (parametri `/`), jolloin kyseiset HTTP-pyynnöt ohjataan lokaalille sovelluspalvelimelle porttiin 8080.

### 5.2.2 Säiliötekniikat ja Docker

*Säiliö* (*container*) on virtuaalikonetta kevyempi, eristetty suoritusympäristö. Toisin kuin perinteiset virtualisointitekniikat, säiliö voi hyödyntää käyttöjärjestelmän tavallista järjestelmäkutsurajapintaa, mikä vähentää säiliön ajamisessa tarvittavia resursseja ja siten mahdollistaa sen, että säiliöitä voi olla suorituksessa enemmän kuin virtuaalikoneita (Turnbull 2014, 7).

*Docker* on Docker, Inc.-yhtiön kehittämä avoimen lähdekoodin alusta, joka automatisoi sovellusten sijoituksen säiliöihin. Se pyrkii tarjoamaan kevyen ja nopean ympäristön sovelluksen suorittamista varten sekä sujuvan työnkulun sovelluskehitykselle. Docker tarvitsee isäntäkoneekseen yhteensopivan Linux-käyttöjärjestelmän, kuten Ubuntun. (Turnbull 2014, 7–8, 18)

Docker-kuva koostuu kerroksista tiedostojärjestelmiä. Alimmassa kerroksessa on käynnistystiedostot (*bootfs*), sen päällä käyttöjärjestelmän peruskuva (*base image*) ja sen yllä sovelluskohtaiset kerrokset, joihin voi liittyä esimerkiksi ohjelmakirjastojen asennuksia. Ajettaessa Docker-kuva syntyy Docker-säiliö, jolloin Docker-kuvan ylimmän kerroksen päälle asetetaan kirjoitettava tiedostojärjestelmä. Jos jotain alemmassa kerroksessa olevaa tiedostoa muokataan, se kopioidaan ylimpään kerrokseen ja muokkaus tapahtuu tässä kerroksessa. Muut kuin ylin kerros ovat siis muuttumattomia. (Turnbull 2014, 73–75)

*Dockerfile* on tiedosto, jonka avulla voidaan rakentaa Docker-kuva. Dockerfilessä kuvataan kuvan rakentamisessa tarvittavat käskyt allekkain, jotka suoritetaan ylhäältä alas. Kukin käsky lisää Docker-kuvaan yhden tiedostojärjestelmäkerroksen. Listauksessa 5.2 on esimerkki Dockerfilestä, joka sisältää viisi käskyä. Käsky `FROM` määrittää käyttöjärjestelmän peruskuvan sekä tarvittaessa version. Käskyllä `RUN` voidaan suorittaa mikä tahansa komento. Esimer-

kissä suoritetaan ensin Ubuntun paketinhallinnan päivityskomento `apt-get update`, minkä jälkeen asennetaan Python 3 (`apt-get install -y python3`). Lopuksi luodaan yksinkertainen Python-komentojono, joka ajettaessa tulostaa `Hello world`. Käsky `CMD` määrittää komennon, joka suoritetaan ajettaessa säiliö.

```
FROM ubuntu:12.04
RUN apt-get update
RUN apt-get install -y python3
RUN echo "print('Hello world')" > script.py
CMD python3 script.py
```

Listaus 5.2. Esimerkki Dockerfilestä.

### 5.2.3 Flask

*Flask* on Python-kielellä toteutettu avoimen lähdekoodin web-sovelluskehys. Se on kevyt mutta helposti laajennettavissa oleva ja soveltuu niin pienien kuin isompienkin websovellusten kehittämiseen (Aggarwal 2014, 1, 7).

Olennainen osa Flaskia ovat reitit (*routes*). Reitti liittää yhteen URL-osoitteen ja funktion, jota kutsutaan, kun reitin URL-osoitteeseen sopiva HTTP-pyyntö saapuu sovellukselle (Grinberg 2014, 8). Tällaista funktiota sanotaan näkymäfunktioksi (*view function*). Yhteen näkymäfunktioon voi liittyä yksi tai useampi reitti. Näkymäfunktion palauttama arvo lähetetään HTTP-vastauksena selaimelle.

Esimerkissä 5.3 on yksinkertainen Flask-sovellus, jossa määritellään yksi reitti ja siihen liittyvä näkymäfunktio `hello_user`. Reitin hyväksymät URL-osoitteet määritellään `@app.route(...)`-dekoraattorilla. Esimerkissä reittiin liittyy parametri `name`, jonka perusteella palautetaan tervehdysteksti. Jos sovellusta kutsutaan esimerkiksi URL-osoitteella, jonka polkuosa on `/hello/world`, niin palautetaan `Hello, world!`.

```
from flask import Flask, Response
app = Flask(__name__)

@app.route('/hello/<name>')
def hello_user(name):
    return Response('Hello, {}!'.format(name), mimetype='text/plain')

if __name__ == '__main__':
    app.run()
```

Listaus 5.3. Yksinkertainen Flask-sovellus.

Esimerkissä 5.3 näkymäfunktio palauttaa `Response`-objektin, jonka sisältö määritellään merkkijonon avulla ja jonka MIME-tyypiksi asetetaan `text/plain`. Näkymäfunktiota voitaisiin muokata niin, että merkkijono vastaa HTML-dokumenttia ja HTTP-vastauksen MIME-tyypiksi asetettaisiin `text/html`. Sama asia on kuitenkin järkevämpi toteuttaa käyttäen Jinja2-muottia (*Jinja2 template*). Se koostuu HTML-tiedostosta ja siinä mahdollisesti olevista lausekkeista, joiden arvot määräytyvät näkymäfunktion kutsuessa muotin koontifunktiota `render_template`. Listauksessa 5.5 on esimerkki muotista ja listauksen 5.4 näkymäfunktiossa kyseinen muotti kootaan antaen parametriksi selaimelta tullut `name`, jolloin lopputuloksena saadaan listauksessa 5.6 esitetty HTML.

```
from flask import Flask
app = Flask(__name__)

@app.route('/hello/<name>')
def hello_user(name):
    return render_template(
    'hello.html', name=name)

if __name__ == '__main__':
    app.run()
```

Listaus 5.4. Jinja2-muotin käyttäminen Flaskissa.

```
<!DOCTYPE html>
<title>Hello, {{name}}!</title>
<h1>Hello, {{name}}!</h1>
```

Listaus 5.5. Jinja2-muotti `hello.html`.

```
<!DOCTYPE html>
<title>Hello, world!</title>
<h1>Hello, world!</h1>
```

Listaus 5.6. Muotti `hello.html` koottuna `name`-parametrin arvolla `world`.

Flask-sovellukseen voidaan määritellä sovellusosioita (*blueprints*), joiden avulla sovellusta voi jakaa pienempiin osiin. Kullekin sovellusosiolle voidaan määritellä reittejä ja näkymäfunktioita. Sovellusosio täytyy rekisteröidä pääsovellusoliolle, jotta sovellusosion reitit saadaan käyttöön. Listauksessa 5.7 on esimerkki sovellusosion `greetings` rekisteröimisestä. Kyseisen sovellusosion määrittely on esitetty listauksessa 5.8. Kutsuttaessa sovellusosion

reittiä sen alkuosaan liitetään sovellusosion nimi. Esimerkiksi näkymäfunktiota `hi_user` vastaavan reitin URL-osoitteen polkuosa on `/greetings/hi/<name>`.

```
from flask import Flask
from greetings import greetings

app = Flask(__name__)
app.register_blueprint(greetings)
if __name__ == '__main__':
    app.run()
```

Listaus 5.7. Sovellusosion rekisteröiminen Flaskissa.

```
from flask import Blueprint

greetings = Blueprint('greetings', __name__)

@greetings.route('/hello/<name>')
def hello_user(name):
    return Response('Hello, {}!'.format(name), mimetype='text/plain')

@greetings.route('/hi/<name>')
def hi_user(name):
    return Response('Hi, {}!'.format(name), mimetype='text/plain')
```

Listaus 5.8. Sovellusosion määritteleminen tiedostossa `greetings.py`.

### 5.2.4 AngularJS

*AngularJS* on Googlen kehittämä avoimen lähdekoodin JavaScript-ohjelmistokehys (Branas 2014, 8; Freeman 2014, 3). Sen perustana on MVC-suunnittelumalli (Freeman 2014, 3), jonka osat ovat seuraavat (Seshadri ja Green 2014, 2):

- *Malli* (*model*) kuvaa sovelluksessa käsiteltyä tietoa,
- *näkymä* (*view*) on käyttöliittymä, joka generoidaan mallin sisältämän tiedon perusteella sekä
- *ohjain* (*controller*) sisältää sovelluslogiikan, joka esimerkiksi hakee tietoa palvelimelta ja päättää, miten ja mitkä osat siitä näytetään.

*Moduulit* ovat AngularJS:n tapa jaotella koodia. Moduuli voi määritellä ohjaimia ja funktioita, jotka ovat käytettävissä kaikkialla moduulissa. Moduuli voi lisäksi riippua muista moduuleista, mikä mahdollistaa koodin uudelleenkäytön. (Seshadri ja Green 2014, 15)

*Palvelut* (*services*) sisältävät toiminnallisuuksia, jotka ovat yleiskäyttöisiä ja jotka eivät kunnolla sovellu sijoitettavaksi suoraan mihinkään MVC-mallin osista. Näitä ovat esimerkiksi sovelluslokitus, tietoturva-asiat sekä verkkotoiminnot. (Freeman 2014, 473)

*Direktiivit* ovat HTML-kielen laajennoksia, jotka tarjoavat uusia toiminnallisuuksia sekä mahdollistavat uudelleenkäytettävien komponenttien luonnin (Branas 2014, 18). Esimerkiksi direktiivi `ng-model="x"` liitettynä `input`-elementin attribuutiksi luo sidoksen `input`-elementissä näkyvän arvon sekä ohjaimen muuttujan `x` välille, jolloin muuttujan `x` arvoa muutettaessa myös käyttöliittymässä näkyvä arvo muuttuu. Vastaavasti käyttäjän muokatessa arvoa käyttöliittymästä myös muuttuja `x` päivittyy.

Listauksessa 5.9 on esimerkki yksinkertaisesta AngularJS-sovelluksesta. Direktiivi `ng-app` määrittää, että kyseessä on AngularJS-sovellus ja että aloitusmoduulin nimi on `exampleApp`. Elementtiin `body` liitetään `ExampleCtrl`-ohjain. Muotti `{{name}}` ilmoittaa, että kyseiseen kohtaan sijoitetaan suorituksen aikana ohjaimen `name`-ominaisuuden arvo – tässä tapauksessa `world`.

Moduuli `exampleApp` ja siihen liittyvä ohjain `ExampleCtrl` määritellään `script`-elementissä. Funktioon `angular.module` menevä toinen parametri `[]` ilmaisee, ettei kyseisellä moduulilla ole riippuvuuksia - muussa tapauksessa ne lueteltaisiin tässä. Metodi `controller` saa parametreinaan ohjaimen nimen sekä taulukon. Kyseinen taulukko sisältää ohjaimen tarvitsemat palvelut sekä funktion, joka saa parametreinaan vastaavat palvelut. Kyseinen funktio määrittelee ohjaimen toiminnallisuuden. Palvelu `$scope` määrittää sen osan tietomallista, joka kyseiselle ohjaimelle on näkyvissä.

```
<!DOCTYPE html>
<html ng-app="exampleApp">
  <head><title>Example AngularJS application</title></head>
  <body ng-controller="ExampleCtrl">
    Hello, {{name}}!.
    <script src="angular.js"></script>
    <script>
      angular.module('exampleApp', [])
      .controller('ExampleCtrl', ['$scope', function($scope) {
        $scope.name = 'world';
      }]);
    </script>
  </body>
</html>
```

Listaus 5.9. Esimerkki AngularJS-sovelluksesta.

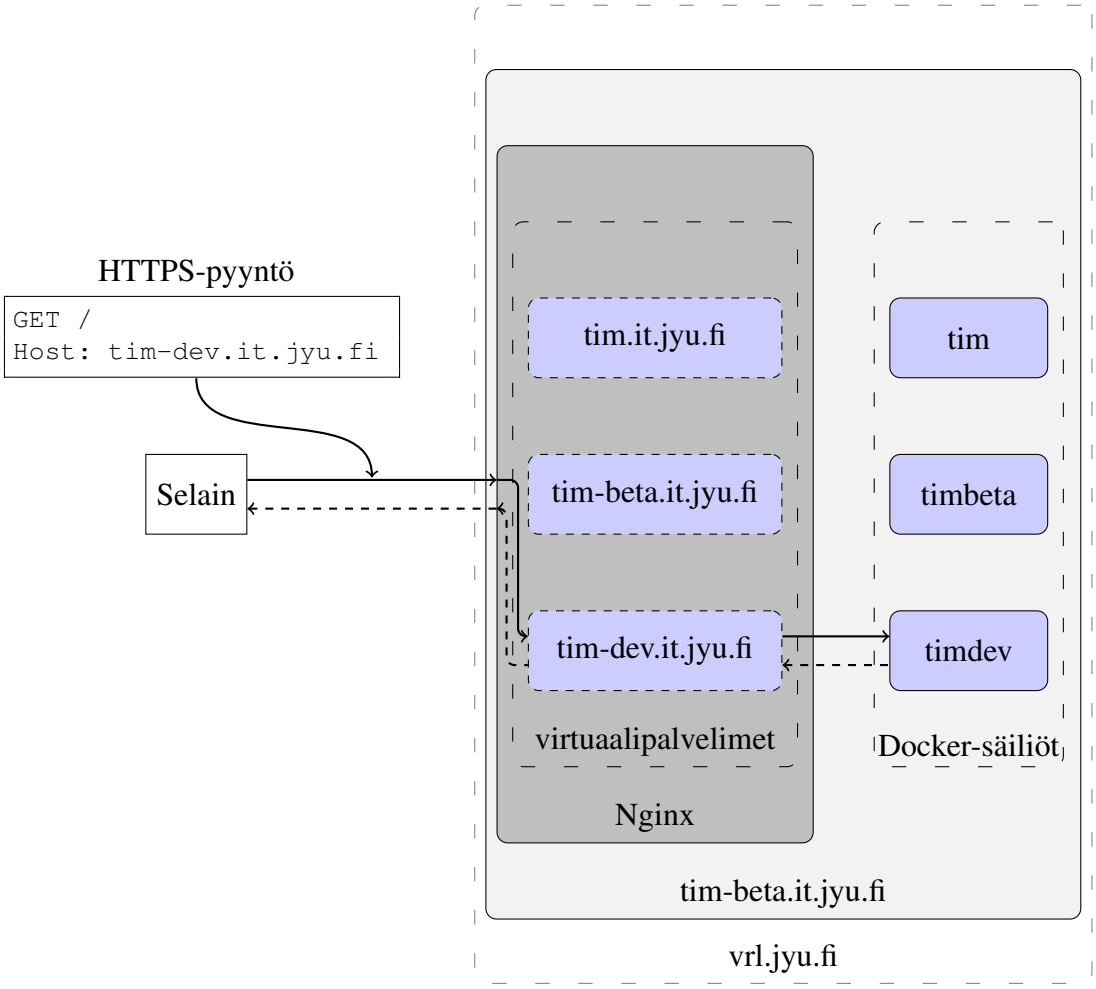
## 5.3 TIM-järjestelmän rakenne

TIM-sovellus on sijoitettu Jyväskylän yliopiston `vrl.jyu.fi`-koneella olevaan virtuaalikoneeseen `tim-beta.it.jyu.fi`, jossa käyttöjärjestelmänä on Ubuntu Linux. Kyseisessä virtuaalikoneessa on Nginx, johon on määritelty seuraavat kolme virtuaalipalvelinta:

- `tim.it.jyu.fi` on tuotantopalvelin, joka on loppukäyttäjien varsinaisessa käytössä,
- `tim-beta.it.jyu.fi`-palvelimella ovat testikäytössä olevat uudet ominaisuudet sekä
- `tim-dev.it.jyu.fi`-palvelin on kehityspalvelin, jossa ominaisuuksia kehitetään ja joka voi olla epästabiili.

Virtuaalikoneeseen tuleva HTTPS-pyyntö ohjataan jollekin edellä mainituista palvelimista sen `Host`-otsikkotietueen perusteella. Jos otsikkotietueen arvona on esimerkiksi `tim-dev.it.jyu.fi`, niin pyyntö ohjataan `tim-dev.it.jyu.fi`-palvelimeen. Jos `Host`-tietueen arvona on pelkkä virtuaalikoneen IP-osoite, niin pyyntö ohjataan tuotantopalvelimeen.

Kutakin virtuaalipalvelinta kohden on ajossa Docker-säiliöt tim, timbeta ja timdev, joissa kussakin on käynnissä Flask-sovellus. Virtuaalipalvelin ohjaa HTTPS-pyynnön vastaavalle säiliölle salaamattomana, joka edelleen ohjaa pyynnön Flask-sovellukselle. Tämä prosessoi HTTP-pyynnön ja palauttaa HTTP-vastauksen, joka säiliön rajapinnan kautta ohjataan takaisin virtuaalipalvelimelle ja edelleen Nginxin toimesta salattuna käyttäjän selaimelle. Ku-




Kuvio 1. Timdev-palvelimelle lähetetyn HTTPS-pyynnön kulku TIM-järjestelmässä.

viossa 1 on esitetty edellä kuvattu timdev-palvelimelle lähetetyn HTTPS-pyynnön kulku.

Kuhunkin Docker-säiliöön (tim, timbeta ja timdev) on liitetty virtuaalikoneen hakemisto (`/opt/tim`, `/opt/tim-beta` ja `/opt/tim-dev` vastaavassa järjestyksessä), jossa sijaitsee TIM-sovelluksen koodivarasto. Säiliön hakemistoon tekemät muutokset näkyvät siis myös säiliön ulkopuolella ja päinvastoin. Näin säiliön sovelluksen päivittäminen uudempaan versioon on sujuvampaa, eivätkä säiliön tekemät muutokset sovelluksen tietokantaan häviä, jos säiliö suljetaan.

## 5.4 TIM-sovelluksen näkymät

TIM-sovelluksessa on viisi päänäkymää: hakemistonäkymä, dokumenttinäkymä, dokumentin hallintanäkymä, asetusnäkymä sekä opettajan näkymä.

### 5.4.1 Hakemistonäkymä

Kuviossa 2 on esitetty TIM-sovelluksen hakemistonäkymä. Tässä näkymässä on listattu taulukossa kansiot ja hakemistot, joihin kirjautuneella käyttäjällä on pääsyoikeus. Dokumentin nimeä napsauttamalla avautuu dokumenttinäkymä, ja kansion nimeä napsauttamalla näkee kyseisessä kansiossa olevat kansiot ja dokumentit. Lisäksi käyttäjä voi luoda uuden dokumentin tai ladata sovellukseen olemassa olevan laitteeltaan.

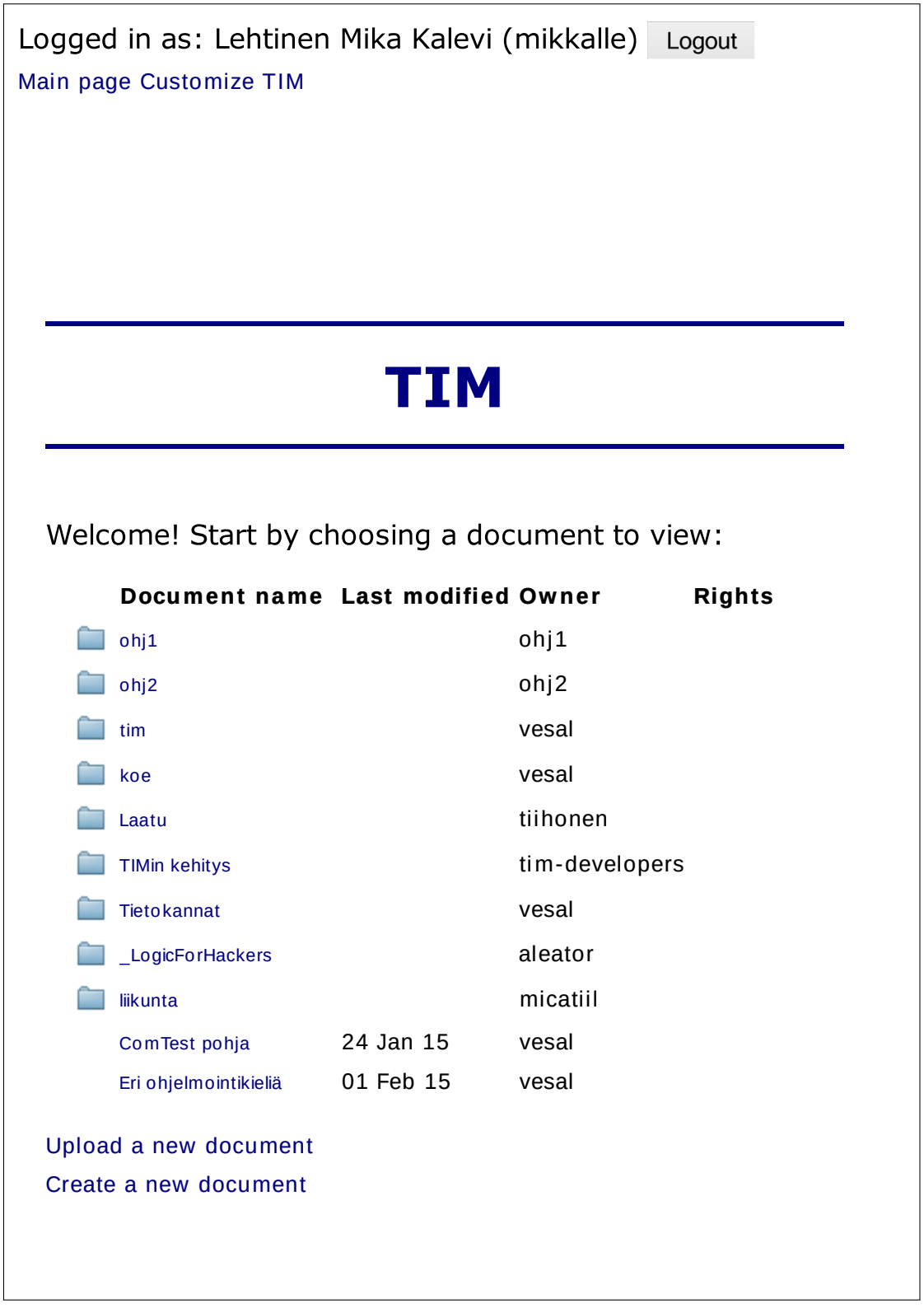
### 5.4.2 Dokumentin hallintanäkymä

Kuviossa 3 on esitetty dokumentin hallintanäkymä. Kyseiseen näkymään pääsevät vain ne kirjautuneet käyttäjät, jotka kuuluvat dokumentin omistajaryhmään. Näkymä tarjoaa seuraavat tiedot ja toiminnot:

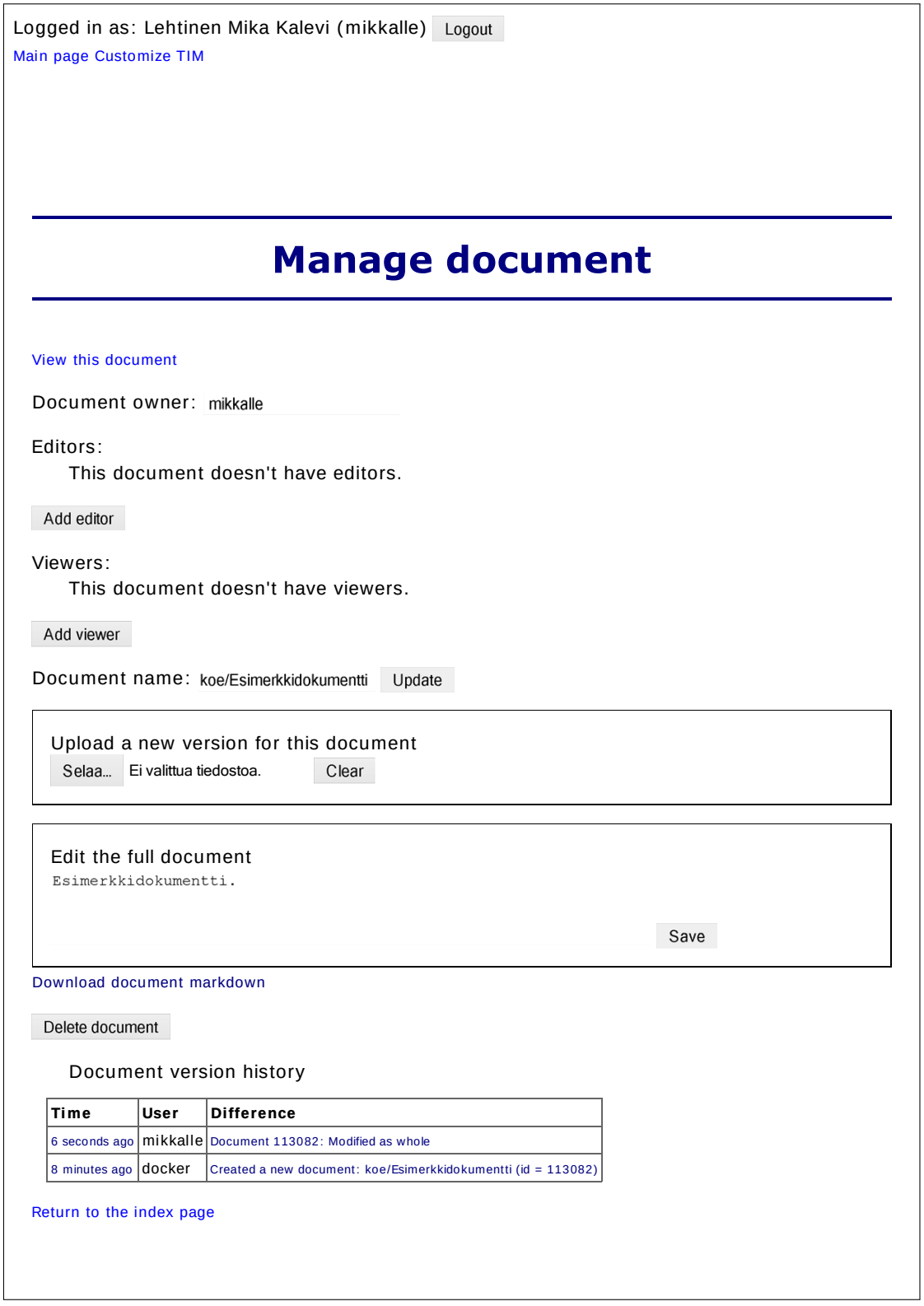
- dokumentin omistajaryhmän vaihtaminen,
- dokumentin muokkaus- tai lukuoikeuden lisääminen tai poistaminen,
- dokumentin poistaminen,
- dokumentin muutoshistoria,
- dokumentin päivitys joko tiedostosta tai tekstilaatikosta,
- dokumentin nimen vaihtaminen,
- dokumentin kahden peräkkäisen version muutosten tarkasteleminen tekstimuodossa sekä
- dokumentin aiemman version tarkasteleminen tekstimuodossa.

### 5.4.3 Asetusnäkymä

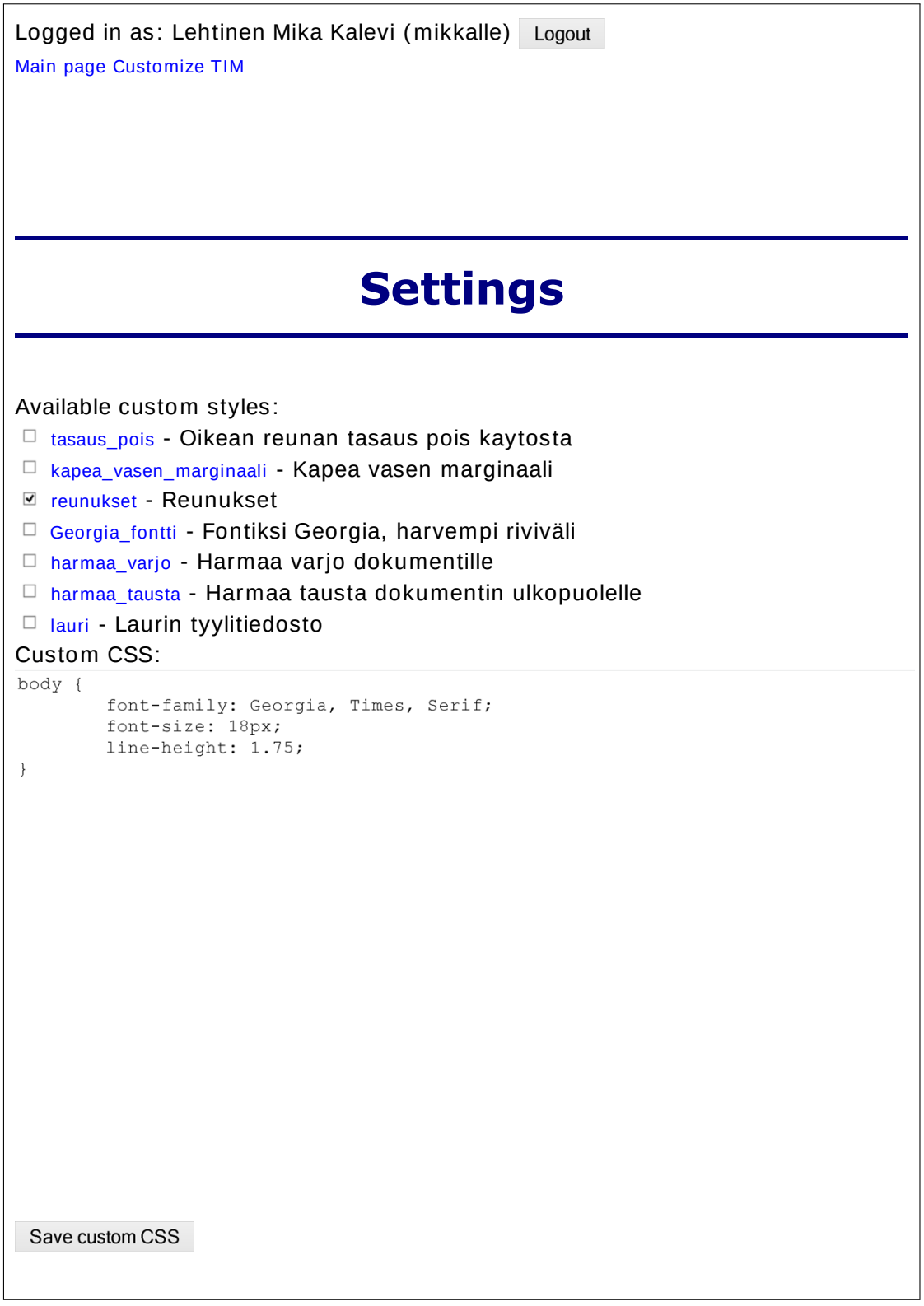
Kuviossa 4 on esitetty asetusnäkymä, jossa käyttäjä voi valita erilaisia CSS-tyylejä käyttöönsä sovellukseen. Käyttäjä voi halutessaan myös kirjoittaa oman CSS-määrityksensä.




Kuvio 2. TIM-sovelluksen hakemistonäkymä.




Kuvio 3. Dokumentin hallintanäkymä.




Kuvio 4. TIM-sovelluksen asetusnäkymä.

### 5.4.4 Dokumenttinäkymä

Kuviossa 6 on esitetty dokumenttinäkymä. Tässä näkymässä käyttäjä voi lukea dokumenttia ja oikeuksista riippuen mahdollisesti myös muokata sitä. Dokumenttiin voi myös lisätä kommentteja, jotka voivat olla joko anonyymejä julkisia tai vain itselle näkyviä. Omia kommentteja voi muokata tai poistaa jälkikäteen, ja dokumentin omistaja voi muokata ja poistaa mitä tahansa kommentteja. Dokumentin oikean laidan punaiset palkit merkitsevät kappaleita, joita käyttäjä ei vielä ole napsauttanut luetuksi. Keltainen palkki merkitsee, että kyseinen kappale on muuttunut viimeisen luetuksi merkitsemisen jälkeen.

### 5.4.5 Opettajan näkymä

Kuviossa 5 on esitetty opettajan näkymä. Tämä näkymä on muuten samanlainen kuin dokumenttinäkymä, mutta opettaja (eli dokumentin omistaja) voi valita, kenen opiskelijan tehtävien vastauksia hän haluaa tarkastella.

## 5.5 TIM-dokumenttien muoto

TIM-dokumentit kirjoitetaan *Pandoc*-merkintäkielellä. Se on laajennettu versio John Gruberin kehittämästä *markdown*-merkintäkielestä (MacFarlane 2015). Yksi markdown-merkintäkielen ominaispiirteistä on se, että sitä voi lukea myös raakatekstimuodossa muuntamatta sitä ensin esimerkiksi HTML-muotoon. Listauksessa 5.10 on esimerkki Pandoc-merkintäkielisestä dokumentista. Jäljempänä termillä *markdown* viitataan nimenomaan Pandoc-merkintäkieleen.

```
# Tämä on otsikko.

Tämä on tavallinen tekstikappale.

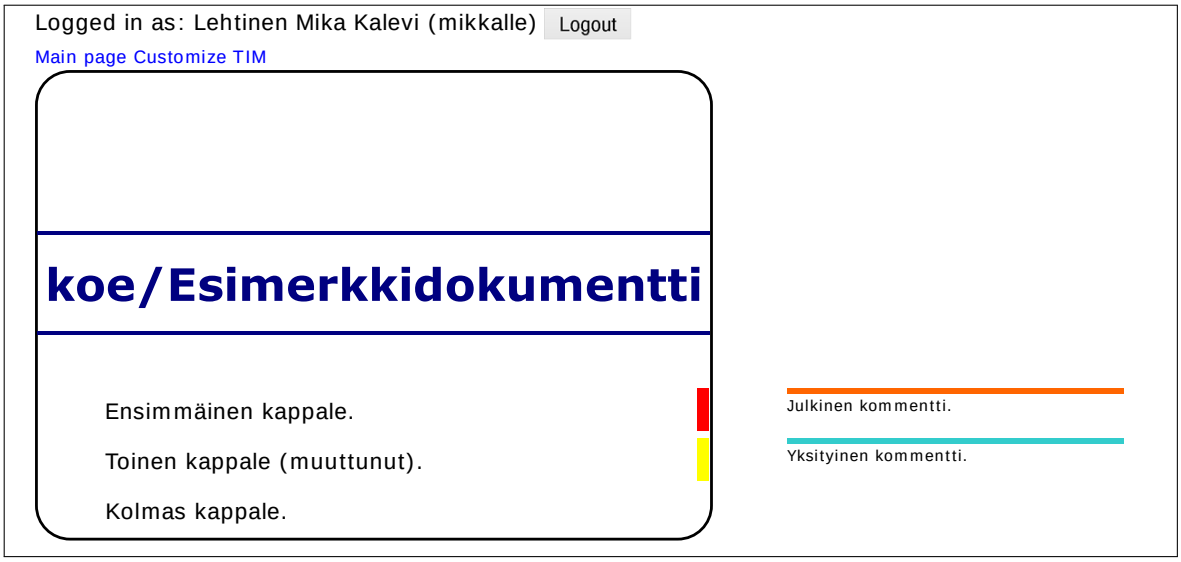
## Tämä on toisen tason otsikko.

* Tästä alkaa luettelo.
* Luettelon toinen alkio.
* Luettelon kolmas alkio.
```

Listaus 5.10. Esimerkki Pandoc-merkintäkielen mukaisesta dokumentista.

TIM-dokumentteihin on mahdollista upottaa liitännäisiä, jotka voivat olla esimerkiksi ajetta-


Kuvio 5. TIM-sovelluksen opettajan näkymä.




Kuvio 6. TIM-sovelluksen dokumenttinäkymä.

via lähdekoodin osia tai monivalintatehtäviä. Liitännäiset merkitään dokumenttiin Pandocin koodilohkomerkinnällä, jolle annetaan attribuuteiksi liitännäisen tyyppi sekä tunniste. Koodilohkon sisään laitetaan liitännäiselle annettavat parametrit. Listauksessa 5.11 on esimerkki monivalinta-tyyppisestä liitännäisestä, jonka tyyppi on `mmcq` ja tunniste `t1`. Kun käyttäjä tallentaa vastauksen tehtävään, siitä tallentuu tietokantaan tehtävätunniste `x.t1`, missä `x` on sen dokumentin tunnistenumero, jossa tehtävä esiintyy.

```
``` {#t1 plugin="mmcq"}
---
stem: "Valitse joko oikein (true) tai väärin (false)."
choices:
  -
    correct: true
    reason: "Vaihtoehto 1 oli oikein"
    text: "Vaihtoehto 1"
  -
    correct: false
    reason: "Vaihtoehto 2 oli väärin"
    text: "Vaihtoehto 2"
```
```

Listaus 5.11. Esimerkki TIM-dokumentin liitännäisestä.

Dokumentin näyttämiseen selaimessa liittyy useampi vaihe. Aluksi markdown muunnetaan HTML-muotoon. Koska markdown-kielen sekaan voi kirjoittaa HTML-kieltä, niin ne kappaleet, jotka eivät ole liitännäisiä, sanitoidaan. Tämä tarkoittaa potentiaalisesti vaarallisten

elementtien poistamista tuotetusta HTML:stä. Sanitointi tapahtuu whitelist-pohjaisesti: tietyt ennalta määritellyt elementit sallitaan ja kaikki muut poistetaan.

Kunkin liitännäiskappaleen kohdalla luetaan liitännäisen tyyppi ja kutsutaan vastaavan liitännäisen rajapintaa antaen parametreiksi liitännäislohkon sisältö. Liitännäinen palauttaa tätä vastaavan HTML-kielen sekä mahdollisen listan tarvittavista AngularJS-moduuleista sekä JS- ja CSS-tiedostoista. Liitännäisen palauttama HTML asetetaan osaksi dokumenttia sen koodilohkon tilalle, josta liitännäisen HTML generoitiin.

# 6 Tutkimusmetodi

Tietojärjestelmätieteen tutkimuksessa keskeisiä paradigmoja ovat käyttäytymistiede (*behavioral science*) sekä suunnittelutiede (*design science*) (Hevner ym. 2004). Kyseisiä paradigmoja voidaan luonnehtia seuraavasti:

- Käyttäytymistiede pyrkii kehittämään ja perustelemaan teorioita, jotka selittävät tai ennustavat inhimillisiä ilmiöitä liittyen tietojärjestelmien analyysiin, suunnitteluun, toteutukseen, hallintaan ja käyttöön (Hevner ym. 2004).
- Suunnittelutiede pyrkii luomaan artefakteja, jotka palvelevat ihmistä (March ja Smith 1995). Artefaktien on tarkoitus ratkaista havaittuja inhimillisiä ja organisatorisia ongelmia mahdollistamalla tietojärjestelmien tehokkaan analyysin, suunnittelun, toteutuksen, hallinnan ja käytön (Hevner ym. 2004).

Käyttäytymis- ja suunnittelutiede ovat komplementaarisia. Käyttäytymistieteen tavoitteena on totuus, kun taas suunnittelutieteen tavoitteena on hyöty. Käyttäytymistieteen tulokset informoivat suunnittelua ja hyöty informoi teoriaa. (Hevner ym. 2004)

Suunnittelutieteen artefaktit voivat olla käsitteitä, malleja, metodeita tai ilmentymiä. Käsitteet tarjoavat kielen, joilla ilmiöitä voidaan kuvata ja jonka avulla ongelmat ja niiden ratkaisut voidaan määritellä. Käsitteiden avulla voidaan muodostaa reaalimaailmaa kuvaavia malleja. Metodit määrittelevät ongelmanratkaisuprosesseja, jotka voivat olla esimerkiksi algoritmeja tai tekstuaalisia kuvauksia. Ilmentymät puolestaan ovat käsitteiden, mallien tai metodien toteutuksia todellisessa järjestelmässä. (Hevner ym. 2004; March ja Smith 1995)

Suunnittelutieteellinen prosessi koostuu kahdesta syklissä suoritettavasta aktiviteetista: artefaktin konstruoinnista ja sen evaluoinnista (March ja Smith 1995). Konstruointivaiheessa artefaktia rakennetaan jonkin määritellyn ongelman ratkaisemiseksi, ja evaluointivaiheessa arvioidaan, kuinka hyvin artefakti soveltuu tähän tarkoitukseen (Hevner ym. 2004).

Tämä tutkielma toteutetaan suunnittelutieteellisenä tutkimuksena. Tutkielmassa tarkastellaan, kuinka TIM-järjestelmässä havaitut yleiset web-sovellusten haavoittuvuudet voidaan korjata. Siten tutkielman tuloksena syntyy joukko metodeja ja/tai ilmentymiä, jotka pyrkivät

antamaan ratkaisun kyseiseen ongelmaan. Kyseisiä artefakteja arvioidaan seuraavilla kriteereillä:

- kuinka tehokkaasti ne ehkäisevät haavoittuvuuksia (esim. jääkö jotain reunatapauksia selvittämättä) sekä
- kuinka helposti ne ovat toteutettavissa.

## 6.1 Tutkimuksen kulku

Tutkimus aloitetaan kartoittamalla TIM-järjestelmän nykyiset haavoittuvuudet. Tämä tehdään suorittamalla sekä black-box- että white-box-analyysi.

Black-box-testauksessa käytettäviä avoimen lähdekoodin ohjelmistoja on olemassa useita, kuten W3AF, Arachni, Zed Attack Proxy sekä IRONWASP. Sekä kaupallisia että avoimen lähdekoodin penetraatiotestaustyökaluja on vertailtu monessa tutkimuksessa, mutta kyseisten tutkimusten tuloksia on vaikea verrata keskenään, sillä ne tutkivat osittain eri työkaluja ja ne käyttävät eri testaustekniikoita (Muñoz ja Villalba 2013).

TIM-sovellus on vahvasti JavaScript-pohjainen. Penetraatiotestaussovelluksen tulee siis pystyä analysoimaan ja suorittamaan JavaScript-koodia. Kyseistä ominaisuutta ei ole W3AF-sovelluksessa (Riancho 2015), joten kyseistä sovellusta ei voida käyttää. Sen sijaan Arachni-sovelluksessa on integroitu selainympäristö, joka mahdollistaa modernien web-sovellusten testaamisen (Laskos 2015a). Penetraatiotestausohjelmistoksi valitaan siis Arachni.

White-box-analyysivaiheessa lähdekoodia tarkastellaan kunkin haavoittuvuuden kannalta tarkoituksenmukaisella tavalla. Esimerkiksi XSS-injektioita tutkittaessa kartoitetaan kohdat, jossa dynaaminen merkkijono sijoitetaan osaksi selaimelle menevää HTML:ää ja tutkitaan, onko merkkijono eskapoitu oikein.

Kukin haavoittuvuustyyppi pyritään ensisijaisesti korjaamaan hyödyntämällä mahdollisia olemassa olevia teknologioita ja kirjastoja (kuten Flask-kehyksen laajennoksia).

# 7 TIM-järjestelmän black-box-testaus

Luvussa kuvataan TIM-järjestelmän black-box-testausprosessi sekä sen tulokset.

## 7.1 Työkalun esittely

*Arachni* on Tasos Laskoksen kehittämä avoimen lähdekoodin web-sovellusten penetraatiotestaustyökalu. Kehittäjän mukaan työkalu kehittää itseään testausprosessin aikana tarkastelemalla web-sovelluksen käyttäytymistä. Se ottaa huomioon web-sovelluksen dynaamisen luonteen ja se voi havaita web-sovelluksessa tapahtuvat muutokset kun työkalu käy läpi web-sovelluksen reittejä. Arachnissa on myös integroitu selainympäristö, jonka avulla kompleksisten, JavaScript-pohjaisten web-sovellusten testaaminen on mahdollista. (Laskos 2015b)

Arachnilla on web-käyttöliittymä, jossa käyttäjä voi määritellä testausprofiileita. Sovellus tarjoaa kolme valmista profiilia (`Default`, `SQL Injection` ja `Cross-Site Scripting`), joista käyttäjä voi halutessaan muokata oman testausprofiilin. Testausprofiili sisältää seuraavia testausprosessiin liittyviä tietoja:

- testauksen laajuuteen liittyvät säännöt,
- minkä tyyppiset syötteet auditoidaan (esim. lomakkeet, evästeet, linkit, otsikkotietueet),
- millaisia merkkijonoja tietynnimisiin lomakekenttiin syötetään,
- millä tavalla työkalu kommunikoi web-sovelluksen kanssa (esim. korkein yhtäaikaisten HTTP-pyyntöjen määrä),
- mitä alustoja työkalun tulisi odottaa (esim. Linux/Windows, Python/PHP),
- mitä haavoittuvuuksia tulisi testata (esim. SQL-injektio, CSRF),
- mitä liitännäisiä halutaan käyttää,
- selainympäristön asetukset (esim. kuvaruudun koko) sekä
- minkä URL-osoitteen ja kaavan mukaan työkalun tulisi tarkastaa istunnon voimassaolo.

Arachni luokittelee löytämänsä ongelmat kolmeen vakavuusluokkaan: korkea (*high*), keski-

taso (*medium*) sekä matala (*low*). Lisäksi Arachni saattaa antaa tiedotteita web-sovelluksen toiminnasta, jotka eivät ole varsinaisia haavoittuvuuksia. Myös löydetyistä haavoittuvuuksista osa voi olla vääriä hälytyksiä, minkä vuoksi kukin täytyy tarkastaa.

## 7.2 Asetukset

TIM-sovelluksen penetraatiotestaukseen käytetään Arachni-sovelluksen viimeisintä kehitysversiota `2.0dev-1.0dev`, sillä viimeisin vakaa versio (`1.0.6`) sisältää ohjelmistovirheen, jonka vuoksi työkalu ei kykene löytämään lainkaan haavoittuvuuksia. Testaamisessa käytetään oletusprofiilia `Default` seuraavin muutoksin:

- Odotettujen alustojen osalta valitaan Linux, SQLite ja Python.
- Istunnontarkastusosoitteeksi valitaan osoite `/checkLogin` ja kaavaksi merkkijono `1`. Kyseinen reitti palauttaa merkkijonon `0` tai `1` riippuen siitä, onko käyttäjä kirjautunut vai ei.
- AutoLogin-liitännäinen otetaan käyttöön, jolle annetaan seuraavat parametrit:
  - Sisäänkirjautumislomakkeen osoite: `/login?emailLogin=1`.
  - Lomakkeen parametrit: `email=arachni@test.com&password=password`.
  - Lauseke, jolla varmistetaan onnistunut sisäänkirjautuminen: `successfully logged in`.
- Otsikkotietueiden auditointi otetaan käyttöön.
- DOM-puun läpikäynnin syvyydeksi asetetaan 20.
- Läpikäytävien URL-osoitteiden joukkoon lisätään seuraavat:
  - `/manage/1`
  - `/view/1`
  - `/teacher/Dokumentti%201`

Profiilin asetukset on kuvattu kokonaisuudessaan liitteessä B YAML-muodossa.

Ennen testaamisen aloitusta TIM-sovellukseen luodaan käyttäjä `arachni@test.com`, jolle luodaan yksi dokumentti, jonka nimi on `Dokumentti 1` ja jonka sisältö on esitetty listauksessa 7.1.

```
# Dokumentti 1

``` {#mmcq plugin="mmcq"}
---
stem: "Esimerkkiliitännäinen"
choices:
  -
    correct: true
    reason: "Syy 1"
    text: "1"
  -
    correct: false
    reason: "Syy 2"
    text: "2"
```
```

Listaus 7.1. Testaamista varten luotavan dokumentin sisältö.

Lisäksi luodaan toinen dokumentti, jonka nimi on `Dokumentti 2` ja jossa on oletussisältö `Edit me!`.

## 7.3 Tulokset

Arachni ilmoitti löytäneensä TIM-sovelluksesta yhteensä viisi haavoittuvuutta, joista kolme on korkean vakavuuden haavoittuvuutta, yksi keskitason sekä yksi matalan tason haavoittuvuus.

Ensimmäinen korkean vakavuuden haavoittuvuus on CSRF-haavoittuvuus dokumentinhallintanäkymässä. Kyseisen näkymän lomakkeissa ei ole lainkaan CSRF-hyökkäyksen ehkäisemisessä tarvittavaa uniikkia tunnistetta, joten haavoittuvuus on todellinen.

Kaksi muuta Arachnin raportoimaa vakavaa haavoittuvuutta ovat samaa tyyppiä: oikeuksienhallinnan sivuutus `Origin`-otsikkotietueen väärennöksellä (*access restriction bypass via origin spoof*). Arachnin mukaan se löysi URL-osoitteen, johon sillä ei ollut pääsyä mutta otsikkotietueen väärennöksen jälkeen se sai oikeuden. Tämä haavoittuvuus on väärä hälytys, sillä TIM-sovellus ei koskaan tarkasta oikeuksia origin-otsikkotietueen perusteella.

Keskitason haavoittuvuudeksi Arachni ilmoitti yleisen hakemiston `/settings`, joka saattaa olla unohdettu ja jota saattaa pystyä hyödyntämään sovelluksen kaappauksessa. Kyseinen sivu on kuitenkin käytössä oleva asetusnäkymä, joten tämä on väärä hälytys.

Matalan tason haavoittuvuudeksi ilmoitettiin puuttuva `X-Frame-Options`-otsikkotietue. Otsikkotiedon puuttuminen mahdollistaa ClickJacking-hyökkäykset, joissa web-sivun sisältö sisällytetään `frame`-elementin avulla hyökkäyssivulle käyttäjän harhauttamiseksi.

Kaiken kaikkiaan Arachni löysi TIM-sovelluksesta siis kaksi haavoittuvuutta: yhden vakavan (CSRF) sekä yhden lievän (puuttuva `X-Frame-Options`-otsikkotietue).

CSRF-haavoittuvuuden korjaaminen käsitellään white-box-analyysin yhteydessä luvussa 8.8. Puuttuva otsikkotietue voidaan helposti lisätä jokaiseen HTTP-vastaukseen hyödyntäen Flask-kehyksen `after_request`-dekoraattoria listauksessa 7.2 kuvatulla tavalla.

```
@app.after_request
def add_x_frame_options(response):
    response.headers.add('X-Frame-Options', 'DENY')
    return response
```

Listaus 7.2. Otsikkotietueen `X-Frame-Options` lisääminen HTTP-vastaukseen Flask-kehyksellä.

# 8 TIM-järjestelmän white-box-analyysi

White-box-analyysissä analysoidaan TIM-järjestelmän koodivaraston puuta `7bcaa8870567bcd429f4308c10ad8c8dffb5334b`, jonka päiväys on 13. maaliskuuta 2015. Analyysistä jätetään pois tiedostot `editing.html` sekä `view.html`, koska ne ovat käytöstä poistuneita muotteja, joita ei ole vielä poistettu koodivarastosta. Myöskään näihin liittyviä JavaScript-tiedostoja hakemistoissa `editView` ja `notesView` ei analysoida. Lisäksi `modules`-hakemistossa olevia liitännäisiä ei analysoida.

## 8.1 Injektio

TIM-järjestelmä tallentaa tietoja SQLite3-tietokantaan sekä tiedostojärjestelmään. Siten TIM-järjestelmä saattaa sisältää seuraavia injektiohaavoittuvuuksia:

- polkuinjektioita,
- käyttöjärjestelmäkomentoinjektioita sekä
- SQL-injektioita.

### 8.1.1 Polkuinjektio

Polkuinjektiossa hakemistopolusta ja tiedostonimestä muodostetaan tiedostopolku siten, että alkuperäinen hakemistopolku ohitetaan kokonaan. Esimerkiksi yhdistettäessä hakemisto `/images` tiedostoon `../../etc/passwd` tiedostopoluksi tulee `/etc/passwd`.

Python-kielessä tiedoston käsittely aloitetaan funktion `open` avulla. Kyseinen funktio saa ensimmäisenä parametrinaan polun siihen tiedostoon, jota käsitellään. Polkuinjektioiden tutkimiseksi riittää siis etsiä ne kohdat, joissa `open`-funktiokutsun ensimmäinen parametri saattaa sisältää käyttäjän syötettä.

Lähdekoodissa `open`-funktiota kutsutaan yhdeksässä eri paikassa. Kutsuista kolme esiintyy metodissa `createDocumentFromBlocks`, joka on poistunut käytöstä eli sitä ei kutsuta enää mistään ja siten se voidaan poistaa kokonaan. Kutsuista yksi on muotoa

```
open(self.getBlockPath(document_id.id), 'rb')
```

missä `document_id.id` on `int`-tyyppinen attribuutti, jonka perusteella `getBlockPath`-metodi muodostaa haettavan dokumentin tiedostopolun. Siten injektiota ei tässä ole.

Kutsuista yksi on muotoa

```
open(os.path.join(self.repo.workdir, path), 'w', newline='\n')
```

joka esiintyy `GitClient`-luokan metodissa `add_custom` saaden parametrinaan merkkijonon `path`. Kyseistä metodia kutsutaan vain yhdestä paikasta vakiomerkkijonoparametreilla, joten tässä tapauksessa injektiota ei synny.

Tietokannan alustusfunktiossa `initializeTables` funktiota `open` kutsutaan muodossa

```
open(schema_file, 'r')
```

missä `schema_file` on parametrina annettu polku tietokannan skeematiedostoon. Funktiota kutsutaan vain vakiomerkkijonoparametrilla `schema2.sql`.

Yksi `open`-kutsuista esiintyy metodissa `writeUtf8` muodossa

```
open(path, 'w', encoding='utf-8', newline='\n')
```

missä `path` on kyseisen metodin yksi parametri. Metodia `writeUtf8` kutsutaan kahdesta paikasta, joissa kummassakaan ei synny injektiota, koska polku muodostetaan yhdistämällä dokumenttien juurihakemisto `blocks_path` sekä dokumentin `int`-tyyppinen tunnistenumero.

Kaksi viimeistä `open`-metodikutsua esiintyvät `Images`-luokan metodeissa `getImage` ja `saveImage`. Näissä kutsuissa ensimmäinen parametri on muotoa

```
self.getImagePath(image_id, image_filename)
```

missä `image_id` on `int`-tyyppinen kuvan tunnistenumero ja `image_filename` kuvan tiedostonimi, jonka sekä `saveImage` että `getImage` saavat parametrinaan.

Metodia `saveImage` kutsutaan näkymäfunktiosta `upload_file`, jossa parametrille

`image_filename` annetaan arvo `doc.filename`. Näkymäfunktiossa suoritetaan sijoitus

```
doc = request.files['file']
```

joten muuttujan `doc.filename` sisältö määräytyy HTTP-pyynnöstä. Siten polkuinjektio voi olla mahdollinen.

Metodia `getImage` kutsutaan samannimisestä näkymäfunktiosta `getImage`, jossa parametrin `image_filename` arvoksi annetaan suoraan URL-osoitteessa oleva merkkijono. Siten polkuinjektio saattaa tässäkin olla mahdollinen.

Edellä todetut mahdolliset polkuinjektiot voidaan korjata käyttämällä Flask-kehyksen tarjoamaa `secure_filename`-funktiota. Se poistaa annetusta tiedostonimestä kaikki sellaiset merkit, jotka saattaisivat aiheuttaa polkuinjektion.

### 8.1.2 Käyttöjärjestelmäkomentoinjektio

Käyttöjärjestelmäkomentoinjektiossa web-sovellus kutsuu ulkoista käyttöjärjestelmän komentoa, johon hyökkääjä pääsee liittämään oman komentonsa. Esimerkiksi web-sovellus voisi suorittaa Python-koodissa kutsun

```
os.system('ls images | grep ' + pattern)
```

joka listaisi `images`-hakemistossa olevat kuvatiedostot, jotka täsmäävät merkkijonon `pattern` mukaiseen `grep`-hakuehtoon. Mikäli `pattern` on käyttäjän syötettä, hyökkääjä voisi syöttää sen arvoksi `; rm -rf --no-preserve-root /`, jolloin komennon suorittaminen yrittäisi poistaa kaikki palvelimella olevat tiedostot.

Python-kielessä käyttöjärjestelmäkomennon suorittaminen on mahdollista seuraavilla tavoilla käyttäen `os`- tai `subprocess`-moduulia:

- `os`-moduulin funktiolla `system` tai `popen`,
- `subprocess`-moduulin luokan `Popen` avulla tai
- `subprocess`-moduulin funktion `call`, `check_call` tai `check_output` avulla.

Näistä ainostaan `subprocess`-moduulin `Popen`-luokka esiintyy TIM-sovelluksen lähdekoo-

dissa, ja sen kaikki parametrit ovat vakioita. Siten sovelluksessa ei ole käyttöjärjestelmäkomentoinjektiohaavoittuvuuksia.

### 8.1.3 SQL-injektio

Kaikki SQL-lauseet suoritetaan tietokantakursoriobjektin kautta `execute`-metodilla. Kyseinen metodi tarvitsee kaksi parametria, joista ensimmäinen on suoritettavan SQL-kyselyn muotti ja toinen on lista kyselyyn sidottavista parametreista. Listauksessa 8.1 on esimerkki SQL-kyselyn suorittamisesta, jossa tietokannasta haetaan käyttäjän tunnistenumero nimen perusteella.

```
cursor = self.db.cursor()
cursor.execute('SELECT id FROM User WHERE name = ?', [name])
result = cursor.fetchone()
```

Listaus 8.1. Esimerkki SQL-kyselystä User-tauluun.

Useimmat TIM-sovelluksen lähdekoodissa esiintyvät SQL-kyselyt ovat muotoa, jossa `execute`-metodin ensimmäinen parametri on vakiomerkkijono. Tällöin SQL-injektion syntyminen on mahdotonta, koska kyseessä on parametrisoitu kysely.

Lähdekoodissa on kuitenkin kahdeksan SQL-kyselyä, joissa `execute`-metodin ensimmäinen parametri, *kyselymuotti*, ei ole vakiomerkkijono. Kyseiset kyselyt on listattu liitteessä A.

Metodeissa `getUsersForTasks` ja `getDocumentsByIds` kyselymuotin parametrimuuttujien lukumäärä riippuu parametrina tuodun listan koosta, ts. kyselymuotissa tulee olemaan yhtä monta kysymysmerkkiä kuin listassa alkioita. SQL-kyselyyn ei siis synny injektiota.

Metodissa `getAnswersForGroup` kyselymuottiin yhdistetään moniosainen ehtolause, jonka kussakin osassa esiintyy käyttäjän tunnistenumero. Tunnistenumero sijoitetaan käyttäen `%d`-muottia, jolloin muun kuin numeerisen arvon liittäminen merkkijonoon on mahdotonta. Siksi SQL-kysely on turvallinen, eikä injektiota synny.

Metodissa `getParMappings` kyselymuottiin loppuun liitetään ehdollisesti (jos `end_index >= 0`) lisäehto `and par_index < {}`, missä lausekkeen `{}` paikalla on muuttujan `end_index` arvo. Koska `end_index` on välttämättä numeerinen arvo (muuten

ehtolause aiheuttaisi esimerkiksi poikkeuksen `unorderable types: str() > int()`), SQL-injektiohaavoittuvuutta ei ole.

Metodin `clear` tarkoitus on tyhjentää tietokannan sisältö. Muuttuja `TABLE_NAMES` on vakiotaulukko, joka sisältää tietokannan taulujen nimet. Kyseistä metodia ei kutsuta minkään reitin kautta, vaan on tarkoitettu kehittäjille.

Metodissa `getMappedValues` on kolme kyselymuottia. Ensimmäisen kyselymuotin arvo määräytyy parametrien `extra_fields`, `table`, `UserGroup_id`, `custom_access` sekä `order_by_sql` perusteella. Näistä `UserGroup_id` voi tulla osaksi kyselyä vasta parametrien sitomisvaiheessa (`execute`-metodikutsu). Metodia `getMappedValues` kutsutaan kolmesta paikasta lähdekoodissa, ja kaikissa kutsuissa edellä mainitut parametrit (pois lukien `UserGroup_id`) ovat vakioita, eli ne eivät ole peräisin käyttäjän syötteestä.

Kahdessa muussa kyselyssä kyselymuotin arvo riippuu ainoastaan parametrin `table` arvosta, joten injektiohaavoittuvuutta ei näissäkään ole.

## 8.2 Viallinen käyttäjän tunnistaminen ja istunnon hallinta

TIM-sovellukseen on mahdollista rekisteröityä kahdella tavalla: sähköpostitse tai Korppi-tunnuksen avulla.

### 8.2.1 Käyttäjän rekisteröityminen ja kirjautuminen Korppi-tunnuksen avulla

Käyttäjä kirjautuu Korppi-tunnuksella napsauttamalla ”Kirjaudu Korpilla”-painiketta. Tällöin TIM generoi 192-bittisen satunnaistunnisteen, joka tallennetaan `appcookie`-istuntomuuttujaan. Tämän jälkeen TIM kutsuu Korpin `allowRemoteLogin`-reittiä, jossa `request`-parametrin arvona on äsken generoitu tunniste. Kyseinen reitti palauttaa tyhjän HTTP-vastauksen, jos kirjautumista ei ole vielä suoritettu. Tällöin TIM ohjaa käyttäjän Korpin kirjautumissivulle antamalla `allowRemoteLogin`-reittiin `authorize`-parametriksi `appcookie`-muuttujan arvon sekä `returnTo`-parametrissa kirjautumisreitin osoitteen.

Käyttäjä kirjautuu Korppiin tavallisesti ja hyväksyy tietojen lähettämisen TIM-sovellukselle, minkä jälkeen Korppi ohjaa käyttäjän selaimen samaiseen TIM-sovelluksen kirjautumisreit-

tiin. Nyt `allowRemoteLogin`-reitti palauttaa käyttäjän tiedot (käyttäjätunnuksen, nimen ja sähköpostiosoitteen), jolloin TIM tarvittaessa rekisteröi tietokantaan uuden käyttäjän Korpista saaduilla tiedoilla ja lopuksi kirjaa käyttäjän sisään.

### 8.2.2 Käyttäjän rekisteröityminen sähköpostitse

Sähköpostirekisteröitymisessä käyttäjää pyydetään antamaan sähköpostiosoite. Jos kyseinen osoite todetaan säännöllisellä lausekkeella `^[\w\.-]+@([\w-]+\.)+[\w-]+$` kelvolliseksi, TIM generoi väliaikaisen salasanan ja lähettää sen tähän sähköpostiosoitteeseen. Samalla TIM pyytää käyttäjää kirjoittamaan kyseisen väliaikaisen salasanan sekä nimen ja uuden salasanan vahvistuksen kera.

Käyttäjän lähetettyä tiedot TIM tarkastaa seuraavat asiat:

1. sähköpostiin lähetetty väliaikainen salasana vastaa käyttäjän syöttämää,
2. kyseisen sähköpostiosoitteen omaavaa Korppi-käyttäjää ei ole vielä olemassa,
3. salasanavahvistus täsmää salasanaan sekä
4. salasanan pituus on vähintään kuusi merkkiä.

Mikäli jokin edellä mainituista kohdista ei toteudu, käyttäjälle näytetään asianmukainen virheilmoitus. Muussa tapauksessa, jos kyseisen sähköpostiosoitteen omaavaa sähköpostikäyttäjää ei ole olemassa, tietokantaan lisätään uusi käyttäjä ja tälle käyttäjäryhmä. Käyttäjän kirjoittamaa salasanaa ei tallenneta tietokantaan sellaisenaan, vaan siitä lasketaan SHA256-tiiviste, joka tallennetaan tietokantaan. Käyttäjätunnukseksi asetetaan sähköpostiosoite.

Jos annetun sähköpostiosoitteen omaava sähköpostikäyttäjä on jo olemassa, päivitetään tämän nimi ja salasana. Tämä käytännössä mahdollistaa siis nimen ja/tai salasanan vaihtamisen.

### 8.2.3 Käyttäjän kirjautuminen sähköpostitse

Sähköpostikirjautumisessa käyttäjältä pyydetään sähköpostiosoite ja salasana. Jos kyseisellä sähköpostiosoitteella ja salasanasta lasketulla SHA256-tiivisteellä löytyy tietokannasta käyttäjä, kirjautuminen hyväksytään. Muussa tapauksessa näytetään virheilmoitus ”käyttäjätun-

nus tai salasana ei täsmää”.

### 8.2.4 Puutteet sähköpostirekisteröitymisessä ja -kirjautumisessa

Sähköpostirekisteröitymisessä ja -kirjautumisessa on havaittavissa seuraavat puutteet:

- Ainoa rajoite käyttäjän määrittämälle salasanalle on kuuden merkin minimipituus. Salasanan laatu tulisi tarkastaa kattavammin esimerkiksi zxcvbn-salasanamittarilla.
- TIM-sovellus ei rajoita kirjautumisyritysten määrää, joten potentiaalinen hyökkääjä voi arvuutella salasanoja (tietäessään käyttäjätunnuksen) rajoittamattomasti.
- Salasanasta tallennetaan tietokantaan suolaamaton SHA256-tiiviste. Salasanan tiivisteen laskemiseksi tulisi käyttää jotakin nimenomaan salasanoja varten suunniteltua tiivistefunktiota.

### 8.2.5 zxcvbn-kirjaston käyttöönotto

zxcvbn on JavaScript-kirjasto, joka määrittelee funktion `zxcvbn`, joka tarvitsee parametrit `password` ja `user_inputs`. Tässä `password` on arvioitava salasana ja `user_inputs` taulukko merkkijonoja, jotka oletetaan mukaan kirjaston käyttämään sisäiseen sanakirjaan, kun salasana arvioidaan. Näitä voivat olla esimerkiksi käyttäjän käyttäjätunnus ja sähköpostiosoite, jolloin kirjasto osaa arvioida salasanan heikoksi, mikäli se muistuttaa käyttäjätunnusta tai sähköpostiosoitetta.

Funktio `zxcvbn` palauttaa joukon salasanasta laskettuja ominaisuuksia, joista käyttäjän kannalta oleellisin on kokonaisluku `score`, joka kuvaa salasanan vahvuutta asteikolla 0–4. Tämä luku voidaan esittää käyttäjälle sanallisesti (esim. ”*Weak*”) ja/tai graafisesti. Listauksissa 8.2 ja 8.3 on esitetty zxcvbn-salasanamittarin käyttäminen AngularJS-kirjaston avulla. Funktiossa `evaluatePassword` tarkastimen sanakirjaan lisätään käyttäjän käyttäjänimi, koko nimi sekä sähköpostiosoite.

```
<label>New password:
    <input name="password"
           class="loginFieldInput"
           type="password"
           ng-model="password"
           ng-change="evaluatePassword()">
</label>
<span>{{ passwordMessage }}</span>
```

Listaus 8.2. zxcvbn-salasanamittarin käyttäminen AngularJS-kirjaston avulla, HTML-osa.

```
signupModule.controller("FormController", ['$scope', function($scope) {
    $scope.userName = {{ session.user_name|tojson }};
    $scope.realName = {{ session.real_name|tojson }};
    $scope.email = {{ session.email|tojson }};
    $scope.evaluatePassword = function() {
        var results = zxcvbn($scope.password,
                             [$scope.userName,
                              $scope.realName,
                              $scope.email]);
        var msgs = ["Very weak", "Weak", "Moderate",
                    "Quite good", "Good!"];
        $scope.passwordMessage = msgs[results.score];
    };
}]);
```

Listaus 8.3. zxcvbn-salasanamittarin käyttäminen AngularJS-kirjaston avulla, JavaScript-osa.

### 8.2.6 bcrypt-kirjaston käyttöönotto

Salasanan tiivisteen muodostaminen TIM-sovelluksessa tapahtuu `hashPassword`-funktiossa, joka on esitetty listauksessa 8.4. Se saa parametrinaan salasanan ja palauttaa vastaavan SHA256-tiivisteen.

Käytettäessä bcrypt-kirjastoa on erotettava erikseen tapaus, jossa salasanan tiiviste lasketaan ensimmäistä kertaa (ts. rekisteröitymisen yhteydessä), sillä tällöin joudutaan generoimaan salasanaan liitettävä suolaosa. Tämä voidaan toteuttaa listauksessa 8.5 esitetyn `generatePasswordHash`-funktion avulla, joka kutsuu `bcrypt`-moduulin `hashpw`-funktiota parametreinaan salasana sekä generoitu suolaosa.

Kirjautumisen yhteydessä tehtävälle salasanan tarkastamiselle voidaan luoda listauksessa 8.6 esitetty funktio `verifyPassword`, joka saa parametreinaan käyttäjän kirjoittaman salasa-

nan sekä tietokantaan aiemmin tallennetun salasanatiivisteen. Kyseiset parametrit välitetään edelleen `hashpw`-funktiolle, jonka palauttaman arvon tulee vastata tietokannassa olevaa tiivistettä.

```
def hashPassword(self, password: 'str') -> 'str':
    return hashlib.sha256(password.encode()).hexdigest()
```

Listaus 8.4. TIM-sovelluksen salasanatiivisteen generointifunktio.

```
def generatePasswordHash(self, password: 'str') -> 'str':
    return bcrypt.hashpw(password, bcrypt.gensalt())
```

Listaus 8.5. Funktion `generatePasswordHash` toteutus.

```
def verifyPassword(self, password: 'str', stored_hash: 'str') -> 'bool':
    return bcrypt.hashpw(password, stored_hash) == stored_hash
```

Listaus 8.6. Funktion `verifyPassword` toteutus.

### 8.2.7 Kirjautumisyritysten rajoittaminen

Kirjautumisyritysten rajoittaminen voidaan toteuttaa lisäämällä käyttäjätauluun `User` sarakkeet `last_failed_login` ja `num_failed_logins`, joista ensimmäinen on viimeisimmän epäonnistuneen kirjautumisyrityksen aikaleima ja jälkimmäinen peräkkäisten epäonnistuneiden kirjautumisyritysten lukumäärä. Tämän lisäksi määritellään vakiot `max_login_attempts` ja `login_wait_time`, joista ensimmäinen kuvaa kirjautumisyritysten enimmäismäärää. Yritysten tullessa täyteen käyttäjä voi yrittää kirjautumista uudelleen muuttujan `login_wait_time` määräämän ajan kuluttua.

Kirjautumisyritysten rajoittamiseen liittyy ongelmia. Jos käyttäjä käyttää kirjautumisyrityksensä kirjoittamalla salasanan useasti väärin mutta kirjoittaakin välittömästi seuraavalla yrityksellä oikein, niin kirjautumisyritystä ei voida hyväksyä odotusajan vuoksi. Lisäksi odotusaika alkaa alusta. Käyttäjälle ei myöskään voida kertoa odotusajan pituutta, sillä muuten hyökkääjä voisi optimoida hyökkäystään järjestelmää vastaan.

Kirjautumisyritysten rajoittamistoimintoa voidaan käyttää myös vandalismiin: tiedossa olevan käyttäjätunnuksen voi lukita syöttämällä jatkuvasti väärän salasanan kirjautumisessa. Vandalismin voisi ehkäistä lukitsemalla tunnuksen käytön ainoastaan tietylle IP-osoitteelle,

mutta tällöin hyökkääjä voisi kiertää rajoituksen käyttämällä esimerkiksi bottiverkkoa.

### 8.2.8 Istunnon hallinta

Flask-kehyksessä on sisäänrakennettu istunnon hallinta. Istunnot ovat ns. asiakaspuolen istuntoja, mikä tarkoittaa sitä, että istunnon tiedot on tallennettu evästeeseen, joka on kryptografisesti allekirjoitettu. Käyttäjä voi siis nähdä evästeestä istuntonsa tiedot muttei muuttaa niitä, sillä tällöin allekirjoitus mitätöityisi.

Istuntoevästeen allekirjoitukseen käytetään Flask-sovelluksen salaista avainta `SECRET_KEY`. Siten turvallinen istunnon hallinta edellyttää, että kyseinen avain on riittävän monimutkainen ja salainen. Tätä konfiguraatioon liittyvää seikkaa käsitellään tarkemmin luvussa 8.5.

## 8.3 Cross-Site Scripting

Kaikki TIM-sovelluksen HTML-dokumentit generoidaan Jinja2-muottien pohjalta. Mahdolliset muuttujat, joista osa voi olla käyttäjän syötteestä peräisin, liitetään osaksi HTML-dokumenttia `{{ param }}`-syntaksilla, missä `param` on muuttujan nimi. Tällöin muuttuja HTML-eskapoidaan automaattisesti (Grinberg 2014, 23). Jos esimerkiksi `param` olisi merkkijono `<b>Hello</b>`, niin muotti muuttaisi sen muotoon `&lt;b&gt;Hello&lt;/b&gt;`.

Jos muottimuuttujaan ei haluta eskapointia, siihen on käytettävä `safe`-suodatinta, joka ottaa eskapoinnin pois käytöstä. Tällöin esimerkiksi lausekkeen `{{ param|safe }}` tuottama merkkijono arvo olisi sama kuin muuttujan `param` arvo.

Heijastettujen ja pysyvien XSS-injektiohaavoittuvuuksien kartoittamiseksi on tutkittava ne kohdat, joissa `safe`-suodatinta käytetään. Näissä tilanteissa on varmistettava, että vastaava muuttuja on joko turvallinen (eli käyttäjä ei voi suoraan vaikuttaa muuttujan arvoon) tai että muuttujasta on karsittu pois haitalliset elementit (kuten `script`) jo aiemmin.

Suodatinta `safe` käytetään muoteissa 12 kertaa, joista kaksi esiintyy HTML-kontekstissa ja loput 10 JavaScript-kontekstissa (eli `script`-elementissä). Näistä 10:stä yhdeksässä `safe`-suodatinta käytetään `tojson`-suodattimen kanssa. Flaskin dokumentaatiossa mainitaankin, että `tojson`-suodattimen kanssa on käytettävä `safe`-suodatinta, mikäli ollaan JavaScript-

kontekstissa. Flaskin versiossa 0.10 `safe`-suodatin onkin automaattisesti käytössä tällaisessa tapauksessa.

JavaScript-kontekstissa viimeisessä tapauksessa on kyse TIM-dokumentin liitännäisten Angular-moduuliriippuvuuksien listaamisesta. Kyseinen koodi on esitetty listauksessa 8.7. Tässä `jsMods` sisältää niiden moduulien nimet, joita Angular-sovellus tulee tarvitsemaan liitännäisten osalta. Moduulien nimet tulevat liitännäisiltä, eli kyseessä on luotettu muuttuja. Koodia voisi kuitenkin huomattavasti yksinkertaistaa asettamalla `checkModules`-funktiolle parametriksi `{{ jsMods|tojson|safe }}`.

```
var modules = checkModules([
    {% for dep in jsMods %}
        "{{ dep|safe }}",
    {% endfor %}
]);
```

Listaus 8.7. Angular-moduuliriippuvuuksien listaamiskoodi.

Ensimmäinen HTML-kontekstissa esiintyvistä `safe`-suodattimista esiintyy kohdassa, jossa TIM-dokumentin kappaleet sisällytetään HTML-dokumenttiin. Kyseiset markdown-kielestä generoidut kappaleet ovat HTML-muodossa, joten `safe`-suodattimen käyttäminen on välttämätöntä, sillä muuten TIM-dokumentti näkyisi selaimessa raakana HTML-kielenä.

Koska merkintäkieleen voi itseensä sisällyttää HTML-elementtejä, on varmistettava, ettei merkintäkielestä generoitu HTML sisällä vaarallisia elementtejä. Tämä varmistetaan siten, että Pandoc $\rightarrow$ HTML-muunnoksen jälkeen HTML sanitoidaan lxml-kirjaston avulla kertomalla, mitkä HTML-elementit ja -attribuutit ovat sallittuja. Jokaista liitännäistä vastaavaa kappaletta kohden kutsutaan kyseisen liitännäisen funktiota, joka generoi kappaletta vastaavan HTML-kappaleen. Liitännäisen palauttama HTML-kappale oletetaan turvalliseksi, eli sitä ei sanitoida, sillä liitännäisen täytyy voida generoida JavaScript-pohjaisia komponentteja.

Toinen HTML-kontekstissa esiintyvistä `safe`-suodattimista esiintyy muotissa, jossa esitetään kahden peräkkäisen dokumentin version välinen ero. Eroa kuvaavan HTML-merkintäkielen laatii funktio `getDifferenceToPrevious`, joka on esitetty listauksessa 8.8. Suodatinta `safe` käytetään siksi, että Git-komennon antama vaste on ANSI-muodossa, jo-

ka muunnetaan HTML-muotoon `ansiconv`-kirjastolla, jotta se näkyisi selaimessa käyttäjäystävällisemmässä muodossa. Koska dokumentin merkintäkieleen on mahdollista kirjoittaa HTML-koodia eikä Git-komennon antamaa vastetta sanitoida, kyseessä on XSS-haavoittuvuus.

Haavoittuvuuden voidaan katsoa johtuvan siitä, ettei `ansiconv`-kirjasto ota HTML-muunnoksessa huomioon, että alkuperäinen teksti saattaa sisältää HTML-merkintäkieltä, jolloin kirjaston vaste sisältää sekä kirjaston generoimia `span`-elementtejä HTML-muunnoksen johdosta että alkuperäisestä tekstistä olevia elementtejä.

Vastaavan toiminnallisuuden omaava kirjasto `ansi2html` ottaa alkuperäisessä tekstissä mahdollisesti esiintyvät HTML-erikoismerkit huomioon, jolloin vasteessa on ainoastaan kirjaston generoimia HTML-elementtejä. Haavoittuvuuden voi siis ehkäistä käyttämällä `ansi2html`-kirjastoa listauksessa 8.9 esitetyn mallin mukaisesti.

```
@contract
def getDifferenceToPrevious(self, document_id: 'DocIdentifier') -> 'str':
    try:
        out, _ = self.git.command('diff --color --unified=5 {}^! {}'
                  .format(document_id.hash,
                          self.getDocumentPathAsRelative(document_id.id)))
    except TimDbException as e:
        e.message = 'The requested revision was not found.'
        raise
    html = ansiconv.to_html(out)
    return html
```

Listaus 8.8. Dokumenttien välisen eron generointikoodi.

```
@contract
def getDifferenceToPrevious(self, document_id: 'DocIdentifier') -> 'str':
    try:
        out, _ = self.git.command('diff --color --unified=5 {}^! {}'
                  .format(document_id.hash,
                          self.getDocumentPathAsRelative(document_id.id)))
    except TimDbException as e:
        e.message = 'The requested revision was not found.'
        raise
    conv = Ansi2HTMLConverter(inline=True, dark_bg=False)
    html = conv.convert(out, full=False)
    return html
```

Listaus 8.9. Dokumenttien välisen eron generointikoodi, jossa XSS-haavoittuvuus on korjattu.

DOM-pohjaisten XSS-haavoittuvuuksien kartoittamiseksi JavaScript-koodista on tutkittava ne kohdat, joissa käyttäjän syötettä liitetään HTML:ksi tulkittuna osaksi dokumentin DOM-puuta. Kun jQuery- ja AngularJS-kirjastot otetaan huomioon, niin kyseisenlainen syötteen liittäminen on mahdollista seuraavilla tavoilla:

- attribuuttien `element.innerHTML` tai `element.outerHTML` avulla, missä `element` on `HTMLElement`-tyyppinen objekti,
- funktioiden `document.write` tai `document.writeln` avulla,
- `eval`-funktion avulla, jolle parametriksi annetaan koodia, jossa esiintyy tällä listalla esiteltyjä tapoja,
- AngularJS:n direktiivin `ng-bind-html` avulla,
- AngularJS:n palvelun `$sce` avulla,
- jQuery-objektin metodien `after`, `before`, `append`, `prepend`, `appendTo`, `prependTo`, `insertAfter`, `insertBefore` tai `add` avulla tai
- jQuery-objektin metodin `html` avulla.

TIM-sovelluksen lähdekoodissa ainoastaan viimeinen tapa on käytössä. Metodit `after`, `before`, `append` ja `prepend` ovat käytössä ainoastaan siltä osin, että niillä liitetään olemassa olevia jQuery-objekteja DOM-rakenteeseen.

jQuery-objektin `html`-funktio ottaa vastaan merkkijonon, joka tulkitaan HTML-kielenä ja muodostuva elementti sijoitetaan jQuery-objektia vastaavan/vastaavien elementtien lapsielementeiksi. Kyseistä funktiota käytetään kolmessa kontekstissa:

- Valittaessa uusi vastaus liitännäisen vastausselaimessa palvelimelta haetaan liitännäistä vastaava HTML-merkintäkieli. Liitännäiseltä tullut HTML on luotettua.
- Kappale-editorin esikatselutoiminto: noin sekunti kirjoittamisen päättämisen jälkeen palvelimelta haetaan kirjoitettua markdown-merkintäkieltä vastaavat HTML-kappaleet, joista ei-liitännäiskappaleet ovat sanitoituja ja liitännäiskappaleet ovat tässäkin tapauksessa luotettuja.
- Tallennettaessa kappale tapahtuu muuten samoin kuin esikatselutoiminnossa, mutta kappaleet sijoitetaan osaksi dokumenttia eikä esikatselualueeseen.

TIM-sovelluksessa ei siis esiinny DOM-pohjaisia XSS-haavoittuvuuksia.

## 8.4 Turvattomat suorat objektiviittaukset

Jokainen TIM-sovellukseen tuleva pyyntö ohjautuu johonkin reittiin tai virhekäsittelijään. Poikkeuksen muodostavat URL-osoitteet, jotka täsmäävät staattisten tiedostojen polkuun. Tämä on oletuksena `/static/`. Osoitteen loppuosa toimii polkuna tiedostolle, jota etsitään staattisten tiedostojen hakemistosta. Tämä on TIM-sovelluksessa `timApp/static`-hakemisto. Kyseinen hakemisto sisältää ainoastaan muuttumattomia CSS-, JS- HTML- ja kuvatiedostoja, joten kaikki viittaukset staattiseen hakemistoon ovat turvallisia.

## 8.5 Turvaton konfiguraatio

TIM-järjestelmästä voidaan havaita kolme eri konfiguraatiokerrosta:

- Nginx-palvelimen konfiguraatio,
- Docker-kuvan konfiguraatio sekä
- Flask-kehyksen konfiguraatio.

Koska Nginx-palvelimen konfiguraatiotiedostoja ei ole versiohallinnassa, tämän käsittely sivuutetaan.

### 8.5.1 Docker-kuvan konfiguraatio

Liitteessä C kuvatussa Dockerfile-tiedostossa voidaan havaita seuraavat konfiguraatioon liittyvät komennot:

- generoidaan lokaali `en_us.utf8`,
- asetetaan globaalisti Git-ohjelmiston käyttäjän sähköpostiosoitteeksi `agent@docker.com` ja nimeksi `agent`,
- luodaan käyttäjä `agent` ja asetetaan tämä omistajaksi `/service`-hakemistoon,
- avataan portit 5000 ja 22,
- asetetaan pääkäyttäjän `root` salasanaksi `test` sekä

- asetetaan aikavyöhykkeeksi `Europe/Helsinki`.

TIM-kuva ajetaan listauksen 8.10 mukaisella komennolla, jossa säiliöön liitetään isäntäkoneen hakemisto `/opt/tim` ja ohjataan portti 50001 säiliön porttiin 5000.

```
docker run --name tim -p 50001:5000 -v /opt/tim/:/service -d -t -i tim-
   new /bin/bash -c 'cd /service/timApp && source initenv.py ; python3
   launch.py ; /bin/bash'
```

Listaus 8.10. TIM-sovelluksen Docker-kuvan ajokomento.

Konfiguraatiokomennoista arveluttavimpia ovat `/service`-hakemiston omistajan vaihtaminen, porttien avaaminen sekä pääkäyttäjän salasanan vaihtaminen. TIM-sovelluksen tiedostosta `tim.py` havaitaan, että Flask-sovellus käynnistetään kuuntelemaan porttia 5000. Porttia 22 puolestaan käytetään ainoastaan kehityksessä, kun halutaan saada SSH-yhteys ajossa olevaan säiliöön. Hakemiston omistajan vaihtaminen ei vaikuta käytännössä mitään, sillä säiliö ajetaan pääkäyttäjänä, ellei toisin käsketä Dockerfilessä tai annettaessa `docker run`-komento.

Pääkäyttäjän salasanan asettaminen triviaaliksi on turvatonta. Lisäksi salasana näkyy joka tapauksessa julkisessa Dockerfilessä, joten salasanan asettaminen monimutkaisemmaksi on turhaa. Toisaalta tieto pääkäyttäjän salasanasta hyödyttää hyökkääjää vain, jos sovelluksessa on olemassa shell-injektiohaavoittuvuus. Vaikka sovelluksen nykyisessä versiossa sellaista ei olisi, kehityksessä sellainen voi syntyä ohjelmointivirheen vuoksi myöhempään versioon. Tällöin hyökkääjä saattaisi kyetä esimerkiksi tuhoamaan sovelluksen tietokannan.

### 8.5.2 Flask-kehyksen konfiguraatio

Flask tarjoaa kolme tapaa konfiguraation lataamiseen: Python-tiedostosta, Python-luokasta tai ympäristömuuttujan osoittaman tiedoston kautta. Näitä vastaavat metodit ovat `from_pyfile`, `from_object` sekä `from_envvar`. Tiedostoon tai luokkaan kirjoitetusta muuttujista ainoastaan isoin kirjaimin kirjoitetut otetaan mukaan konfiguraatioon. Metodien `from_pyfile` ja `from_envvar` totuusarvoparametri `silent` ilmoittaa, halutaanko jatkaa sovelluksen suoritusta heittämättä poikkeusta, vaikka tiedostoa ei löytyisi.

TIM-sovellus lataa oletuskonfiguraation versiohallinnassa olevasta tiedostosta

`defaultconfig.py` ja välittömästi tämän jälkeen ympäristömuuttujan `TIM_SETTINGS` osoittamasta tiedostosta. Kyseinen konfiguraation lataaminen on esitetty listauksessa 8.11.

```
app = Flask(__name__)
app.config.from_pyfile('defaultconfig.py', silent=False)
app.config.from_envvar('TIM_SETTINGS', silent=True)
```

Listaus 8.11. TIM-sovelluksen konfiguraation lataaminen.

Listauksessa 8.10 esitetystä TIM-sovelluksen ajokomennosta voidaan havaita, ettei `TIM_SETTINGS`-muuttujaa käytetä. Siten konfiguraatio määräytyy täysin tiedoston `defaultconfig.py` sisällön perusteella.

Tiedoston `defaultconfig.py` sisältö on esitetty listauksessa 8.12. Tiedostossa määritellään 15 muuttujaa, jotka voidaan jakaa kolmeen eri kategoriaan: Flask-kehyksen sisäänrakennettuihin muuttujiin, Flask-kehyksen laajennosten käyttämiin muuttujiin sekä TIM-sovelluksen omiin muuttujiin. Kunkin muuttujan kategoria, merkitys sekä se, onko muuttuja lainkaan käytössä, on esitetty taulukossa 1. Sellaiset muuttujat, jotka eivät ole käytössä, voidaan poistaa tarpeettomina.

Koska tiedosto `defaultconfig.py` on versiohallinnassa, muuttujan `SECRET_KEY` arvo paljastuu, mitä ei saisi tapahtua. Flask allekirjoittaa istuntoon liittyvän evästeen kyseisellä avaimella, joten tietämällä salausavaimen käyttäjä voi väärentää istuntonsa tietoja mielivaltaisesti.

Erillinen salausavain voidaan lisätä versioimattomaan tiedostoon, joka sijoitetaan tuotantokoneeseen esimerkiksi tiedostoon `secret.py`. Tällöin TIM-säiliön käynnistyskomennossa voidaan määrittää `TIM_SETTINGS`-muuttujan arvoksi kyseinen tiedosto, jolloin salausavain ladataan tästä tiedostosta ylikirjoittaen tiedostossa `defaultconfig.py` oleva arvo.

| Nimi | Kategoria | Merkitys | Käytössä? |
|---|---|---|---|
| COMPRESS_DEBUG | laajennos | onko HTTP-vastausten pakkaus käytössä debug-tilassa | kyllä |
| COMPRESS_MIMETYPES | laajennos | mitkä MIME-tyypit pakataan | kyllä |
| COMPRESS_MIN_SIZE | laajennos | vähimmäiskoko HTTP-vastaukselle, jotta se pakataan | kyllä |
| DEBUG | Flask | onko debug-tila päällä | kyllä |
| MAX_CONTENT_LENGTH | Flask | HTTP-pyynnön enimmäiskoko | kyllä |
| PROFILE | Flask | onko suorituskyvyn mittaus päällä | kyllä |
| SECRET_KEY | Flask | salainen avain | kyllä |
| DATABASE | TIM | polku tietokantatiedostoon | kyllä |
| FILES_PATH | TIM | polku tietokantahakemistoon | kyllä |
| LOG_DIR | TIM | polku lokihakemistoon | kyllä |
| LOG_FILE | TIM | lokitiedoston nimi | kyllä |
| LOG_PATH | TIM | polku lokitiedostoon | kyllä |
| PASSWORD | TIM | salasana | ei |
| UPLOAD_FOLDER | TIM | ladattavien tiedostojen hakemisto | kyllä |
| USERNAME | TIM | käyttäjänimi | ei |

Taulukko 1. TIM-sovelluksen konfiguraatiomuuttujat, niiden tyypit, merkitykset ja käyttötilat.

```
import os

COMPRESS_DEBUG     = True
COMPRESS_MIMETYPES = ['text/html', 'text/css', 'text/xml', 'application/
    json', 'application/javascript']
COMPRESS_MIN_SIZE  = 50
DATABASE           = './tim_files/tim.db'
DEBUG              = False
FILES_PATH         = 'tim_files'
LOG_DIR            = "../tim_logs/"
LOG_FILE           = "timLog.log"
LOG_PATH           = os.path.join(LOG_DIR, LOG_FILE)
MAX_CONTENT_LENGTH = 16 * 1024 * 1025
PASSWORD           = '4t95MHJj9h89y'
PROFILE            = False
SECRET_KEY         = '85db8764yhfZz7-U.-y968buyn89b54y8y45tg'
UPLOAD_FOLDER      = "./media/images/"
USERNAME           = 'admin'
```

Listaus 8.12. Tiedoston `defaultconfig.py` sisältö.

## 8.6 Sensitiivisen tiedon paljastuminen

TIM-järjestelmään tallennetaan käyttäjästä seuraavia tietoja:

- käyttäjätunnus,
- salasanan SHA256-tiiviste,
- lukumerkinnät,
- kommentit,
- tehtävien vastaukset,
- sähköpostiosoite sekä
- globaalit CSS-tyylimääritykset dokumenteille.

Luvussa 3.3.1 todettiin, että salasanan tiivisteen laskemiseksi ei kannata käyttää SHA256-funktiota vaan jotain erityisesti salasanoja varten suunniteltua tiivistefunktiota. Ongelman ratkaisemiseksi `bcrypt`-kirjaston käyttöönottaminen esiteltiin luvussa 8.2.6.

Muista tiedoista potentiaalisesti sensitiivisiä ovat käyttäjätunnus, lukumerkinnät, kommentit ja sähköpostiosoite. Lisäksi jotkin järjestelmään tallennetut dokumentit voivat olla potentiaalisesti sensitiivisiä.

Käyttäjäkohtaiset tiedot tallennetaan salaamattomaan SQLite-tietokantaan, joka sijaitsee TIM-palvelinkoneella. Dokumenttien markdown-tiedostot tallennetaan palvelinkoneen tiedostojärjestelmään salaamattomana. Sen sijaan selaimen ja TIM-palvelimen välinen yhteys on salattu. Tämän tutkielman puitteissa ei oteta kantaa siihen, kuinka mahdollisia sensitiivisiä tietoja voitaisiin suojata paremmin.

## 8.7 Puuttuva ominaisuustason oikeushallinta

TIM-sovelluksessa kullekin dokumentille voidaan määritellä ne käyttäjäryhmät, joilla kyseiseen dokumenttiin on lukuoikeus (DR). Vastaavasti voidaan määritellä muokkausoikeus (DW). Lisäksi jokaisella dokumentilla on omistajaryhmä (DO).

Lukuoikeuden omaavat käyttäjät voivat

- avata dokumentin luettavaksi HTML-muodossa,
- merkitä dokumentin kappaleita luetuiksi,
- jättää kommentteja dokumenttiin sekä
- ajaa dokumentissa olevia tehtäviä, jolloin kyseisen käyttäjän vastaus tallentuu tietokantaan.

On huomattava, etteivät lukuoikeuden omaavat käyttäjät voi tarkastella dokumentin markdown-lähdekoodia, koska liitännäiskappaleet saattavat sisältää tehtävän vastauksen.

Muokkausoikeuden omaavilla käyttäjillä on lukuoikeus ja lisäksi he voivat

- muokata dokumenttia sekä
- tarkastella dokumentin markdown-lähdekoodia.

Dokumentin omistajaryhmään kuuluvilla käyttäjillä on muokkausoikeus ja lisäksi he voivat

- antaa muokkaus- tai lukuoikeuden jollekin käyttäjäryhmälle,
- poistaa muokkaus- tai lukuoikeuden joltakin käyttäjäryhmältä,
- poistaa dokumentin,
- muokata tai poistaa dokumentin kommentteja,
- vaihtaa dokumentin omistajaryhmää sellaiseksi, johon käyttäjä itse kuuluu sekä

- tarkastella käyttäjien antamia vastauksia dokumentissa esiintyviin tehtäviin opettajan näkymässä.

Dokumentin luonti, resurssin lataus palvelimelle ja asetusnäkymään pääseminen edellyttävät, että käyttäjä on kirjautunut (L). Lisäksi kommentin lähettäjä voi muokata tai poistaa kommenttinsa (CW). Kuvan katselemiseen on olemassa erillinen oikeus (IR).

Taulukossa 2 on esitetty TIM-sovelluksen reitit, niiden merkitykset sekä ne vaaditut oikeudet, jotka reitin tulisi tarkastaa edellä mainittujen määritysten mukaisesti. Lisäksi taulukon viimeiseen sarakkeeseen on merkitty ne oikeudet, jotka sovelluksen lähdekoodissa tarkastetaan. Kahden viimeisen sarakkeen arvojen tulisi siis olla samat.

Taulukossa merkintä DR (DO) tarkoittaa sitä, että kyseistä reittiä voi kutsua vastaavan dokumentin omistaja tai käyttäjä, joka hakee omia tietojaan. Esimerkiksi reittiä `/answers/1.t1/2` voi kutsua dokumentin 1 omistaja tai käyttäjä, jonka tunnistenumero on 2.

Voidaan havaita, että reiteissä `/getBlock/<int:docId>/<int:blockId>`, `/getJSON/<int:docId>/<int:blockId>` sekä `/<plugintype>/<task_id>/answer/` oikeuksien tarkastus on virheellinen. Kahdessa ensimmäisessä tarkastetaan ainoastaan lukuoikeus, vaikka reitit palauttavat dokumentin kappaleen markdown-muodossa. Siten lukija voi saada selville koko dokumentin markdown-muodossa kutsumalla jompaakumpaa reittiä useasti muodossa `/reitti/doc_id/n`, missä `doc_id` on dokumentin tunniste ja $n$ käy läpi kokonaisluvut välillä $[0,k[$, missä $k$ on dokumentissa olevien kappaleiden lukumäärä.

Reitistä `/<plugintype>/<task_id>/answer/` puuttuu oikeuksien tarkastus kokonaan, vaikka vain lukijaoikeuden omaavilla käyttäjillä tulisi olla oikeus lähettää dokumentissa oleviin tehtäviin vastauksia.

| **Tyyppi** | **Reitti** | **Merkitys** | **Vaaditut oikeudet** | **Tarkastetut oikeudet** |
|---|---|---|---|---|
| DEL. | /documents/<int:doc_id> | dokumentin poistaminen | DO | DO |

| | | | | |
|---|---|---|---|---|
| GET | /<plugin>/<path:fileName> | liitännäisen resurssin (esim. JavaScript-tiedosto) hakeminen | - | - |
| GET | /answers/<task_id>/<user> | käyttäjän vastausten hakeminen tiettyyn tehtävään | DR (DO) | DR (DO) |
| GET | /diff/<int:doc_id>/ <doc_hash> | dokumentin eron näyttäminen edelliseen | DW | DW |
| GET | /doc/<path:doc_name> | dokumentin näyttäminen | DR | DR |
| GET | /download/<int:doc_id>/ <doc_hash> | dokumentin tietyn version lähdekoodin näyttäminen | DW | DW |
| GET | /download/<int:doc_id> | dokumentin lähdekoodin näyttäminen | DW | DW |
| GET | /getBlock/<int:docId>/ <int:blockId> | dokumentin tietyn kappaleen lähdekoodin hakeminen | DW | DR (!) |
| GET | /getBlockHtml/<int:docId>/ <int:blockId> | dokumentin tietyn kappaleen HTML-muodon hakeminen | DR | DR |
| GET | /getDocuments | käyttäjälle näkyvien dokumenttien metatietojen hakeminen | - | - |
| GET | /getJSON-HTML/ <int:doc_id> | dokumentin hakeminen HTML-formaatissa JSON-listana | DR | DR |
| GET | /getJSON/<int:doc_id>/ | dokumentin lähdekoodin hakeminen JSON-listana | DW | DR (!) |
| GET | /getPermissions/<int:doc_id> | dokumentin oikeustietojen hakeminen | DO | DO |
| GET | /getState | liitännäiskappaleen HTML-formaatin hakeminen tiettyyn tilaan ja käyttäjään liittyen | DR (DO) | DR (DO) |
| GET | /images/<int:image_id>/ <image_filename> | kuvan hakeminen | IR | IR |

| | | | | |
|---|---|---|---|---|
| GET | /images | listaus niiden kuvien tunnisteista, joihin käyttäjällä on katseluoikeus | - | - |
| GET | /index/<int:docId> | dokumentin sisällysluettelon hakeminen | DR | DR |
| GET | /korppiLogin | Korppi-kirjautuminen | - | - |
| GET | /login | kirjautuminen | - | - |
| GET | /manage/<int:doc_id> | dokumentin hallintanäkymä | DO | DO |
| GET | /notes/<int:doc_id> | dokumentissa näkyvien muistiinpanojen hakeminen | DR | DR |
| GET | /quickLogin/<username> | kirjautumisreitti kehittäjille | D | D |
| GET | /read/<int:doc_id> | dokumentin lukumerkintöjen hakeminen käyttäjälle | DR | DR |
| GET | /settings/ | asetusnäkymä | L | L |
| GET | /static/<path:filename> | staattisten resurssien polku | - | - |
| GET | /static/scripts/bower_components/<path:filename> | kolmannen osapuolen JS-tiedostojen polku | - | - |
| GET | /teacher/<path:doc_name> | opettajan näkymä | DO | DO |
| GET | /testuser/<path:anything> | käytöstä poistunut testausreitti | - | - |
| GET | /testuser | käytöstä poistunut testausreitti | - | - |
| GET | /uploads/<filename> | ladatun tiedoston hakeminen | - | - |
| GET | /view/<path:doc_name> | sama kuin /doc/ | | |
| GET | /view_html/<path:doc_name> | sama kuin /doc/ | | |
| GET | / | hakemistonäkymä | - | - |
| POST | /altlogin | kirjautuminen sähköpostilla | - | - |
| POST | /altsignup2 | rekisteröitymistietojen lähetys (askel 2) | - | - |
| POST | /altsignup | rekisteröitymistietojen lähetys | - | - |
| POST | /createDocument | uuden dokumentin luonti | L | L |
| POST | /deleteNote | kommentin poisto | CW/DO | CW/DO |

| | | | | |
|---|---|---|---|---|
| POST | /deleteParagraph/ <int:doc_id>/<int:blockId> | kappaleen poisto | DW | DW |
| POST | /editNote | kommentin muokkaus | CW/DO | CW/DO |
| POST | /log/ | ei käytössä | | |
| POST | /logout | uloskirjautuminen | - | - |
| POST | /newParagraph/ | uuden kappaleen lisääminen | DW | DW |
| POST | /postNote | kommentin lähetys | DR | DR |
| POST | /postParagraph/ | kappaleen muokkaus | DW | DW |
| POST | /preview/<int:doc_id> | kappaleen esikatselu | - | - |
| POST | /settings/save | asetuksien tallennus | L | L |
| POST | /update/<int:doc_id>/ <version> | dokumentin päivittäminen kokonaisena | DW | DW |
| POST | /upload/ | kuvan tai dokumentin lataus | L | L |
| PUT | /<plugintype>/<task_id>/ answer/ | vastauksen lähettäminen liitännäiseen | DR | - (!) |
| PUT | /addPermission/<int:doc_id>/ <group_name>/<perm_type> | oikeuden lisääminen | DO | DO |
| PUT | /changeOwner/<int:doc_id>/ <int:new_owner> | dokumentin omistajan vaihto | DO | DO |
| PUT | /read/<int:doc_id>/ <int:specifier> | lukumerkinnän asettaminen kappaleelle | DR | DR |
| PUT | /removePermission/ <int:doc_id>/<int:group_id>/ <perm_type> | oikeuden poistaminen | DO | DO |
| PUT | /rename/<int:doc_id> | dokumentin uudelleennimeäminen | DO | DO |

Taulukko 2: TIM-sovelluksen reitit, niiden merkitykset sekä vaaditut ja tarkastetut oikeudet.

Oikeuksien tarkastus tapahtuu funktioiden `verifyEditAccess`, `verifyViewAccess` sekä `verifyOwnership` avulla. Jos näkymäfunktio vaatii esimerkiksi dokumentin muokkausoikeuden, niin näkymäfunktion alussa kutsutaan `verifyEditAccess(doc_id)`, missä `doc_id` on parametrina annettu dokumentin tunniste. Tällaisessa tavassa on kuitenkin ainakin seuraavat heikkoudet:

- Oikeuksientarkastusfunktiokutsu on muun lähdekoodin seassa, jolloin ei ole välittö-

mästi selvää, onko tietyssä näkymäfunktiossa olemassa oikeuksientarkastus vai ei. Funktiokutsu ei monesti voi olla näkymäfunktion ensimmäinen lause, sillä tarvittava parametri täytyy ensin hakea HTTP-pyynnöstä.

- Tietyn toimintokokonaisuuden (esim. dokumentin muokkaaminen) näkymäfunktioihin täytyy jokaiseen erikseen lisätä sama oikeuksientarkastuskutsu. Tämä lisää todennäköisyyttä sille, että ainakin yhdestä näkymäfunktiosta oikeuksientarkastus unohtuu.

Oikeuksienhallinta voidaan toteuttaa keskitetymmin siten, että kullekin toimintokokonaisuudelle (kuten dokumentin muokkaaminen) luodaan sovellusosio. Tällöin kyseiselle sovellusosiolle voidaan `before_request`-dekoraattorin avulla määritellä *alustusfunktio*, joka suoritetaan jokaiselle sellaiselle HTTP-pyynnölle, joka osuu johonkin kyseiseen sovellusosioon rekisteröidyistä reiteistä. Tässä aliohjelmassa voidaan suorittaa oikeuksien tarkastus, jolloin tätä ei tarvitse tehdä missään kyseisen sovellusosion näkymäfunktioista.

Alustusfunktion toimiminen edellyttää, että oikeuksien tarkastukseen liittyvät parametrit ovat aina tiettyä muotoa. Esimerkiksi dokumentin muokkaamiseen liittyvissä reiteissä muokattavan dokumentin tunnistenumero tulisi olla tietynnimisessä parametrissa, kuten `doc_id`.

Listauksessa 8.13 on esitetty toteutus alustusfunktiosta, joka tarkastaa, onko käyttäjällä oikeus muokata dokumenttia. Tarvittava parametri `doc_id` voi sijaita HTTP-pyynnön URL-osoitteen polkuosassa, kyselyosassa tai sisältöosan JSON-oliossa. Parametrin haun suorittaa funktio `get_int_param_from_req`. Jos parametria ei löydy, se esiintyy kahdessa tai useammassa paikassa tai sitä vastaavaa dokumenttia ei ole olemassa, palautetaan virheilmoitus. Lopuksi suoritetaan varsinainen muokkausoikeuden tarkastus.

```
@edit_page.before_request
def check_edit_right():
    param_name = 'doc_id'
    doc_id = get_int_param_from_req(param_name)
    timdb = getTimDb()
    if not timdb.documents.documentExists(doc_id):
        abort(404)
    verifyEditAccess(doc_id)

def get_int_param_from_req(param_name):
    doc_id = None
    doc_ids = (request.view_args.get(param_name),
               request.args.get(param_name),
               request.get_json().get(param_name)
               if request.method in ('POST', 'PUT', 'DELETE') else None)
    for d in doc_ids:
        if d:
            if doc_id:
                abort(400, 'Duplicate parameter: {}'.format(param_name))
            doc_id = d
    if not doc_id:
        abort(400, 'Missing {} parameter'.format(param_name))
    doc_id = int(doc_id)
    return doc_id
```

Listaus 8.13. Toteutus muokkausoikeuden tarkastamisesta.

Lukuoikeuden ja omistajuuden tarkastus voidaan toteuttaa vastaavasti hyödyntäen listauksen 8.13 funktiota `get_int_param_from_req`. Sisäänkirjautuneisuuden tarkastamiseksi alustusfunktioksi riittää kutsu `verifyLoggedIn()`.

Oikeuksien tarkastuksen vaativia reittejä on yhteensä 36, joista 30 on toteutettavissa alustusfunktioita ja sovellusosioita käyttäen. Kyseiset 30 reittiä jakautuvat seuraavasti:

- 8 reittiä vaatii dokumentin omistajuuden,
- 9 reittiä vaatii dokumentin lukuoikeuden,
- 9 reittiä vaatii dokumentin muokkausoikeuden sekä
- 4 reittiä vaatii kirjautumisen.

Seuraavat 5 reittiä vaativat monimutkaisemman oikeuksientarkastuslogiikan:

- Kommentin poisto (`/deleteNote`) ja muokkaus (`/editNote`) edellyttävät joko dokumentin omistajuutta tai kommentin omistajuutta.
- Liitännäiskappaleen HTML-formaatin hakeminen tiettyyn tilaan ja käyttäjään liittyen

(`/getState`) edellyttää, että käyttäjä on dokumentin omistaja tai käyttäjä hakee omia tietojaan.

- Käyttäjän vastausten hakeminen tiettyyn tehtävään (`/answers`) edellyttää, että käyttäjä on dokumentin omistaja tai käyttäjä hakee omia tietojaan.
- Reitti `/quickLogin` on tarkoitettu kehittäjille. Sen avulla voi kirjautua sisään minä tahansa käyttäjänä. Reittiin on kovakoodattu kehittäjien käyttäjätunnukset, joilla on oikeus kutsua reittiä.

Lisäksi kuvan lukuoikeuden tarkastaminen on käytössä vain yhdessä reitissä, minkä vuoksi sovellusosion tekeminen kuvan lukuoikeutta varten on toistaiseksi tarpeetonta.

## 8.8 Cross-Site Request Forgery

On helppo havaita, että TIM-sovelluksen kaikki DELETE-, POST- ja PUT-tyyppiset reitit (ts. kaikki sovelluksen tilaa muuttavat reitit) ovat haavoittuvia CSRF-hyökkäykselle, sillä niiden parametrit ovat täysin ennustettavissa. Taulukossa 3 on listattu kyseiset reitit parametreineen ja tyyppeineen. Lisäksi sarakkeessa *Kutsun muoto* on mainittu, millä tavalla reittiä kutsutaan selaimesta.

*Flask-WTF* on Flaskin laajennos, joka tarjoaa automaattisen CSRF-suojauksen kaikille näkymäfunktioille. Suojaus voidaan ottaa käyttöön antamalla sovellusolio parametriksi `CsrfProtect`-funktiolle listauksen 8.14 osoittamalla tavalla.

```
app = Flask(__name__)
CsrfProtect(app)
```

Listaus 8.14. CSRF-suojauksen käyttöön ottaminen Flask-WTF-kirjastolla.

Oletuksena Flask-WTF suojaa metodityypit POST, PUT ja PATCH, joten näiden joukkoon on lisättävä myös DELETE. Tämä tapahtuu lisäämällä konfiguraatiotiedostoon seuraava rivi:

```
WTF_CSRF_METHODS = ['POST', 'PUT', 'PATCH', 'DELETE']
```

Suojaamisen jälkeen kaikissa POST-, PUT-, PATCH- ja DELETE-tyyppisissä metodeissa varmistetaan, että pyyntö sisältää `X-CSRFToken`-nimisen otsikkotietueen ja että sen arvona on voimassa oleva CSRF-tunniste.

| Tyyppi | Reitti | Vaaditut parametrit | Kutsun muoto |
|---|---|---|---|
| DELETE | /documents/<doc_id> | (ei ole) | AJAX (`$http`) |
| POST | /altlogin | email, password | lomake |
| POST | /altsignup2 | realname, token, password, passconfirm | lomake |
| POST | /altsignup | email | lomake |
| POST | /createDocument | doc_name | AJAX (`$http`) |
| POST | /deleteNote | docId, par, note_index | AJAX (`$http`) |
| POST | /deleteParagraph/ <doc_id>/<blockId> | (ei ole) | AJAX (`$http`) |
| POST | /editNote | docId, par, text, access, note_index, tags | AJAX (`$http`) |
| POST | /log/ | message, level | (reittiä ei kutsuta) |
| POST | /logout | (ei ole) | AJAX (`$http`) |
| POST | /newParagraph/ | text, docId, par | AJAX (`$http`) |
| POST | /postNote | docId, par, text, access, tags | AJAX (`$http`) |
| POST | /postParagraph/ | docId, text, par | AJAX (`$http`) |
| POST | /preview/<doc_id> | text | AJAX (`$http`) |
| POST | /settings/save | (ei ole) | AJAX (`$http`) |
| POST | /update/<doc_id>/ <version> | file *tai* fulltext | AJAX (`$http`) |
| POST | /upload/ | file | AJAX (`$http`) |
| PUT | /<plugintype>/ <task_id>/answer/ | input | AJAX (`$http`) |
| PUT | /addPermission/<doc_id>/ <group_name>/<perm_type> | (ei ole) | AJAX (`$http`) |
| PUT | /changeOwner/<doc_id>/ <new_owner> | (ei ole) | AJAX (`$http`) |
| PUT | /read/<doc_id>/<specifier> | (ei ole) | AJAX (`$http`) |
| PUT | /removePermission/<doc_id>/ <group_id>/<perm_type> | (ei ole) | AJAX (`$http`) |
| PUT | /rename/<doc_id> | new_name | AJAX (`$http`) |

Taulukko 3. TIM-sovelluksen DELETE-, POST- ja PUT-reitit sekä niiden parametrit.

Suojaamisen lisäksi on varmistettava, että CSRF-tunniste lähetetään selaimelta jokaisen CSRF-suojatun pyynnön mukana otsikkotietueena. Jos tunnistetta ei ole pyynnön mukana tai se on väärä, palvelin lähettää HTTP-vastauksen tilakoodilla 400, jonka sisältönä on ”CSRF token missing or incorrect”.

AngularJS:n `$http`-palvelu lähettää jokaisen pyynnön mukana `X-XSRF-TOKEN`-nimisen otsikkotietueen, jos eväste `XSRF-TOKEN` on olemassa. Flask voi asettaa kyseisen evästeen kuhunkin HTTP-vastaukseen dekoraattorin `@app.after_request` avulla listauksen 8.15 esittämällä tavalla.

```
@app.after_request
def after_request(resp):
    resp.set_cookie('XSRF-TOKEN', generate_csrf())
    return resp
```

Listaus 8.15. CSRF-tunnisteen asettaminen evästeeseen.

Flask-WTF-kirjasto etsii CSRF-tunnistetta oletusarvoisesti otsikoista `X-CSRFToken` ja `X-CSRF-Token`, jotka poikkeavat AngularJS:n lähettämästä nimestä. Odotetun tietueen nimen voi vaihtaa AngularJS:ää vastaavaksi lisäämällä konfiguraatiotiedostoon seuraavan rivin:

```
WTF_CSRF_HEADERS = ['X-XSRF-TOKEN']
```

Koska kolmea POST-tyyppistä reittiä kutsutaan lomake-elementin kautta, vastaaviin lomakkeisiin on sijoitettava näkymätön kenttä nimeltä `csrf_token` listauksen 8.16 osoittamalla tavalla, jotta palvelin hyväksyy lomakkeen lähetyksen.

```
<input type="hidden" name="csrf_token" value="{{ csrf_token() }}" />
```

Listaus 8.16. CSRF-tunnisteen sijoittaminen lomakkeeseen.

## 8.9 Haavoittuvien komponenttien käyttäminen

TIM-sovellus on ajossa Docker-säiliössä, jonka levykuva kootaan Dockerfile-tiedostossa määritellyillä komennoilla. Kyseisen tiedoston sisältö on listattu liitteessä C. Levykuvan rakentaminen aloitetaan Ubuntun levykuvasta. Koska versiota ei määritetä, käytetään oletuksena Ubuntun uusinta Docker-rekisterissä olevaa versiota. Komento `apt-get update` päivit-

tää saatavilla olevien pakettien metatiedot uusimpiin versioihin. Tämän jälkeen komennolla `apt-get install` asennetaan paketit `python3`, `python3-pip`, `git-core`, `zlib1g-dev`, `libxml2-dev`, `libyaml-dev`, `wget`, `cmake`, `libffi-dev`, `npm` ja `openssh-server`. Komentojen yhteydessä ei mainita versionumeroa, joten näistä paketeista asennetaan uusimmat saatavilla olevat versiot. Näiden lisäksi kirjastosta `libgit2` asennetaan versio 0.22.0 lataamalla lähdekoodipaketti ja kääntämällä se.

Komennolla `pip3 install` asennetaan paketit `flask`, `flask-compress`, `beautifulsoup4`, `pycontracts`, `hypothesis`, `gitpylib`, `lxml`, `pyaml`, `ansiconv` ja `cssutils`. Myös näistä paketeista asentuvat viimeisimmät versiot, sillä versionumeroa ei erikseen määrätä.

Komennolla `bower --allow-root install` asennetaan käytetyt JavaScript-kirjastot, joita ovat `angular`, `angular-ui-ace`, `ng-file-upload`, `jquery`, `angular-sanitize`, `bootstrap` ja `waypoints`. Näissä asennuksissa määrätään versionumero. Syntaksi `#~x.x` tarkoittaa, että kirjastosta asennetaan uusin `x.x`-alkuinen versio. Sen sijaan `angular-ui-ace`-paketin numerosarja `36844ff7c0e0d9445bc8e31514d7f0f59cb8b048` viittaa tiettyyn täsmälliseen versioon.

Voidaan olettaa, että sellaiset komponentit, joista asennetaan uusin versio, eivät todennäköisesti sisällä tunnettuja haavoittuvuuksia. Sen sijaan komponentit, joista määrätään asennettavaksi jokin tietty versio, sisältävät todennäköisemmin haavoittuvuuden, ellei komponentti satu olemaan edelleen uusin versio. Taulukossa 4 on listattu ne komponentit, joista asennetaan tietty versio. Voidaan havaita, että asennetuista komponenteista `ng-file-upload`, `angular-sanitize` sekä `libgit2` ovat vanhentuneita. Ne on siten syytä päivittää uusimpaan versioon.

## 8.10 Tarkastamattomat uudelleenohjaukset

TIM-sovelluksessa on käytössä kahdentyyppisiä uudelleenohjauksia: Flask-kehyksen `redirect`-kutsuja sekä JavaScript-ohjauksia (eli attribuutin `document.location` avulla toteutettuja).

Funktio `redirect` ottaa parametrinaan sen URL-osoitteen, jonne käyttäjä halutaan uudellee-

| Nimi | Tyyppi | Asennettu versio | Uusin versio |
|---|---|---|---|
| angular | JS-kirjasto | ~1.3 | 1.4.3 (1.3.17) |
| angular-ui-ace | Angular-moduuli | 36844ff7 | 36844ff7 |
| ng-file-upload | Angular-moduuli | ~3.0 | 5.0.9 |
| jquery | JS-kirjasto | ~2.1 | 2.1.4 |
| angular-sanitize | Angular-moduuli | ~1.3 | 1.4.4 |
| bootstrap | CSS-kirjasto | ~3.3 | 3.3.5 |
| waypoints | JS-kirjasto | ~3.1 | 3.1.1 |
| libgit2 | C-kirjasto | 0.22.0 | 0.23.0 |

Taulukko 4. TIM-sovelluksen ajoympäristön komponentit, joista on asennettu tietty versio.

nohjata. Kyseisiä funktiokutsuja on TIM-sovelluksessa kahdeksan kappaletta. Näistä kolme on muotoa `redirect(url_for('view_function'))`, missä `view_function` on sen näkymäfunktion nimi, johon käyttäjä halutaan ohjata. Siten ohjausosoite on aina vakio, eikä näihin liity riskiä.

Kutsuista kolme on muotoa `redirect(session.get('came_from', '/'))`, eli ohjausosoite haetaan istuntoon tallennetusta `came_from`-muuttujasta. Kyseinen istuntomuuttuja asetetaan `saveCameFrom`-funktiossa, josta havaitaan, että muuttujan arvo on peräisin HTTP-pyynnön `came_from`-parametrista. Muuttuja tallennetaan istuntoon sellaisenaan, eli käyttäjän on mahdollista asettaa muuttujan `came_from` arvoksi mielivaltainen merkkijono.

Loput kaksi `redirect`-kutsua ovat erilaisia ja siten vaativat oman tarkastelunsa. Näistä ensimmäinen on muotoa

```
redirect(url + "?authorize=" + session['appcookie']
             + "&returnTo=" + urlfile, code=303)
```

jossa `url` on aiemmin aliohjelmassa määritelty vakiomerkkijono, `session['appcookie']` on aiemmin generoitu 24 merkkiä pitkä heksamerkkijono sekä `urlfile`-muuttujan arvona on `request.url_root + "korppiLogin"`. Muuttuja `request.url_root` ilmoittaa palvelimen juuriosoitteen, joka voi olla esimerkiksi `https://tim.it.jyu.fi/`. Siten ohjaukseen ei pääse käyttäjän syötettä.

Viimeinen `redirect`-kutsu on muotoa

```
redirect(url_for('uploaded_file', filename=filename))
```

jossa `filename` on `secure_filename`-funktiosta saatu merkkijono. Funktion `uploaded_file` reitti on muotoa `/uploads/<filename>`, joten ohjaus sivuston ulkopuoliseen osoitteeseen ei ole mahdollista.

Sovelluksessa esiintyy myös yksi JavaScript-ohjaus muotissa `loginpage.html`. Ohjaukseen liittyvä JavaScript-koodi on esitetty listauksessa 8.17.

```
$(document).ready(function () {
    var target_url = {{ target_url|tojson }};
    var separator = target_url.indexOf('?') >= 0 ? '&' : '?';
    var came_from_raw = {{ came_from|tojson }};
    var came_from = encodeURIComponent(came_from_raw.replace("#", "%23"));
    var anchor_raw = window.location.hash.replace('#', '');
    var anchor = encodeURIComponent(anchor_raw);
    $("#anchor").val(anchor_raw);
    window.korppiLogin = function () {
        window.location.replace(target_url
        + separator
        + 'came_from='
        + came_from
        + '&anchor='
        + anchor);
    }
});
```

Listaus 8.17. JavaScript-ohjaus `loginpage.html`-muotissa.

Kutsu `window.location.replace` suorittaa ohjauksen. Ohjausosoite riippuu muuttujista `target_url`, `separator`, `came_from` sekä `anchor`. Muuttujat `target_url` ja `came_from` ovat muottimuuttujia, joiden arvot asetetaan listauksessa 8.18 esitetyssä koontifunktiokutsussa. Tässä muuttuja `target_url` on vakio ja muuttujan `came_from` arvona on nykyisen reitin URL-osoite. Lopullinen URL-osoite on siis muotoa

```
http://tim.it.jyu.fi/korppiLogin?came_from=<came_from>&anchor=<anchor>
```

missä `<came_from>` ja `<anchor>` ovat vastaavan nimisten muuttujien arvot. Ohjaus ei siis voi viedä TIM-sovelluksen ulkopuoliseen osoitteeseen.

```
render_template('loginpage.html',
                target_url=url_for('login_page.loginWithKorppi'),
                came_from=request.url)
```

Listaus 8.18. Muotin `loginpage.html` koontifunktiokutsu.

Koska minkään uudelleenohjauksen tarkoituksena ei ole viedä käyttäjää ulkopuoliselle sivustolle, uudelleenohjauksissa voidaan käyttää aliohjelmaa, joka tarkastaa, onko kohdeosoitteella sama palvelinosoite kuin sovelluksella. Jos palvelinosoitteet täsmäävät, ohjaus hyväksytään. Muussa tapauksessa käyttäjä voidaan esimerkiksi ohjata hakemistonäkymään virheilmoitus näyttäen. TIM-sovelluksen `redirect`-kutsut voidaan korvata listauksessa 8.19 esitetyllä funktiolla `safe_redirect`. Apufunktio `is_safe_url` tarkastaa, onko annettu osoite turvallinen. Osoite todetaan turvalliseksi vain, jos sen protokolla on joko `http` tai `https` ja jos sen palvelinosa on sama kuin sovelluksella.

```
def is_safe_url(url):
    host_url = urlparse(request.host_url)
    test_url = urlparse(urljoin(request.host_url, url))
    return test_url.scheme in ['http', 'https'] and \
           host_url.netloc == test_url.netloc

def safe_redirect(url, **values):
    if is_safe_url(url):
        return redirect(url, **values)
    return redirect(url_for('indexPage'))
```

Listaus 8.19. Funktion `safe_redirect` toteutus.

# 9 Yhteenveto

Tutkielmassa selvitettiin, mitä OWASP Top 10 -listan web-sovellusten haavoittuvuuksia TIM-järjestelmässä on ja miten ne voidaan järkevästi korjata. Selvitys tapahtui suorittamalla järjestelmälle black-box- ja white-box-analyysit sekä hyödyntämällä järjestelmän käytössä olevien tekniikoiden ominaisuuksia.

Black-box-analyysissä havaittiin CSRF-haavoittuvuus sekä puuttuva `X-Frame-Options`-otsikkotietue. Puuttuva otsikkotietue on helposti korjattavissa Flask-kehyksen `after_request`-dekoraattorilla.

White-box-analyysissä löydettiin seuraavat haavoittuvuudet: CSRF-haavoittuvuus, haavoittuva uudelleenohjaus, lievästi puutteellinen käyttäjän tunnistaminen, XSS-haavoittuvuus, turvaton konfiguraatio, puutteellinen ominaisuustason oikeushallinta sekä potentiaalisesti haavoittuvien komponenttien käyttäminen. Seuraavia haavoittuvuustyyppejä ei todettu olevan: injektio, istunnon hallinnan haavoittuvuus sekä turvaton suora objektiviittaus.

Sovelluksessa esiintyneet CSRF-, ja XSS- sekä uudelleenohjaushaavoittuvuudet voidaan ehkäistä hyvin automatisoidusti Flask- ja AngularJS-kehyksien avulla. Lisäksi Flaskin sovellusosioilla voidaan yksinkertaistaa ominaisuustason oikeushallintaa, jolloin ainoastaan monimutkaisemmat oikeushallintatapaukset joudutaan ohjelmoimaan erikseen.

Käyttäjän tunnistamisen osalta todettiin, että sovellus sallii heikkoja salasanoja ja että salasanaan käytettävä tiivistefunktio on heikko. Nämä voidaan korjata käyttäen zxcvbn- ja bcrypt-kirjastoja, joista ensimmäinen tarkastaa salasanan vahvuuden ja jälkimmäinen luo salasanasta vahvan tiivisteen.

Järjestelmän konfiguraatiota tarkastellessa havaittiin, että Flask-kehyksen salainen avain `SECRET_KEY` on julkinen. Tämä mahdollistaa esimerkiksi istuntomuuttujien mielivaltaisen muokkaamisen käyttäjän osalta.

Docker-alustan avulla TIM-säiliön komponentit on helppoa pitää ajan tasalla, jolloin haavoittuvien komponenttien olemassaolo on epätodennäköisempää. Automaation parantamiseksi säiliön käynnistyksen yhteyteen voisi luoda mekanismin, jolla tarkistetaan, onko jolle-

kin komponentille saatavilla päivitys. Mikäli näin on, kehittäjille lähetettäisiin ilmoitus.

Löydetyistä haavoittuvuuksista kriittisimmät eli puutteellinen oikeushallinta, XSS-haavoittuvuus ja turvaton konfiguraatio on korjattu TIM-järjestelmän uusimpaan versioon (marraskuu 2015). Myös muut korjaukset tullaan liittämään uusimpaan versioon riittävän testaamisen jälkeen.

Tutkielmassa onnistuttiin löytämään TIM-järjestelmästä useita haavoittuvuuksia ja antamaan niille käyttökelpoiset korjausehdotukset. Tutkielmassa käsiteltiin kuitenkin vain OWASP-organisaation laatiman Top 10 -listan haavoittuvuuksia, joista turvaton konfiguraatio sekä sensitiivisen tiedon paljastus käsiteltiin vaillinaisesti. Lisäksi liitännäisten tietoturvaa ei analysoitu. Olisi esimerkiksi varmistettava, ettei käyttäjän liitännäiseen syöttämä vastaus voisi koskaan aiheuttaa XSS-injektiota vastauksen lukijalle. Edellä mainitut seikat on syytä huomioida järjestelmän jatkokehityksessä, jolle tämä tutkielma antaa hyvän pohjan.

# Lähteet


Aggarwal, Shalabh. 2014. *Flask Framework Cookbook.* Packt Publishing Ltd.

Aivaliotis, Dimitri. 2013. *Mastering Nginx.* Packt Publishing Ltd.

Antunes, N., ja M. Vieira. 2012. "Defending against Web Application Vulnerabilities". *Computer* 45 (2): 66–72. doi:10.1109/MC.2011.259.

———. 2014. "Penetration Testing for Web Services". *Computer* 47, numero 2 (helmikuu): 30–36. doi:10.1109/MC.2013.409.

Austin, Andrew, Casper Holmgreen ja Laurie Williams. 2013. "A comparison of the efficiency and effectiveness of vulnerability discovery techniques". *Information and Software Technology* 55 (7): 1279–1288. doi:10.1016/j.infsof.2012.11.007.

Barth, A. 2011a. *HTTP State Management Mechanism.* Verkossa; viitattu 17. marraskuuta 2015. https://tools.ietf.org/html/rfc6265.

———. 2011b. *The Web Origin Concept.* Verkossa; viitattu 17. marraskuuta 2015. https://tools.ietf.org/html/rfc6454.

Barth, Adam, Collin Jackson ja John C. Mitchell. 2008. "Robust defenses for cross-site request forgery". Teoksessa *Proceedings of the 15th ACM conference on Computer and communications security,* 75–88. CCS '08. Alexandria, Virginia, USA: ACM. doi:10.1145/1455770.1455782.

Berners-Lee, T., R. Fielding ja L. Masinter. 2005. *Uniform Resource Identifier (URI): Generic Syntax.* Verkossa; viitattu 17. marraskuuta 2015. https://tools.ietf.org/html/rfc3986.

Branas, Rodrigo. 2014. *AngularJS Essentials.* Packt Publishing Ltd.

Carnavalet, Xavier De Carné De, ja Mohammad Mannan. 2015. "A Large-Scale Evaluation of High-Impact Password Strength Meters". *ACM Trans. Inf. Syst. Secur.* (New York, NY, USA) 18, numero 1 (toukokuu): 1:1–1:32. doi:10.1145/2739044.

Doupé, Adam, Marco Cova ja Giovanni Vigna. 2010. “Why Johnny Can’t Pentest: An Analysis of Black-box Web Vulnerability Scanners”. Teoksessa *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment,* 111–131. DIMVA’10. Bonn, Germany: Springer-Verlag. doi:10.1007/978-3-642-14215-4_7.

Dukes, L., Xiaohong Yuan ja F. Akowuah. 2013. “A case study on web application security testing with tools and manual testing”. Teoksessa *Southeastcon, 2013 Proceedings of IEEE,* 1–6. Huhtikuu. doi:10.1109/SECON.2013.6567420.

Durumeric, Zakir, James Kasten, Michael Bailey ja J. Alex Halderman. 2013. “Analysis of the HTTPS Certificate Ecosystem”. Teoksessa *Proceedings of the 2013 Conference on Internet Measurement Conference,* 291–304. IMC ’13. Barcelona, Spain: ACM. doi:10.1145/2504730.2504755.

Al-Fedaghi, Sabh. 2012. “Privacy as a Base for Confidentiality”. *SSRN Electronic Journal.* doi:10.2139/ssrn.2012395.

Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach ja T. Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1.* Verkossa; viitattu 17. marraskuuta 2015. https://tools.ietf.org/html/rfc2616.

Fielding, Roy, ja Julian Reschke. 2014a. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.* Verkossa; viitattu 17. marraskuuta 2015. https://tools.ietf.org/html/rfc7230.

———. 2014b. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.* Verkossa; viitattu 17. marraskuuta 2015. https://tools.ietf.org/html/rfc7231.

Finifter, Matthew, ja David Wagner. 2011. “Exploring the Relationship Between Web Application Development Tools and Security”. Teoksessa *Proceedings of the 2Nd USENIX Conference on Web Application Development,* 9–9. WebApps’11. Portland, OR: USENIX Association. http://dl.acm.org/citation.cfm?id=2002168.2002177.

Fonseca, J., N. Seixas, M. Vieira ja H. Madeira. 2014. “Analysis of Field Data on Web Security Vulnerabilities”. *Dependable and Secure Computing, IEEE Transactions on* 11, numero 2 (maaliskuu): 89–100. doi:10.1109/TDSC.2013.37.

Freeman, Adam. 2014. *Pro AngularJS.* 1st. Berkely, CA, USA: Apress. doi:10.1007/978-1-4302-6449-1.

Grinberg, Miguel. 2014. *Flask Web Development: Developing Web Applications with Python.* 1st. O’Reilly Media, Inc.

Halfond, William G.J., Jeremy Viegas ja Alessandro Orso. 2006. “A Classification of SQL-Injection Attacks and Countermeasures”. Teoksessa *Proceedings of the IEEE International Symposium on Secure Software Engineering,* 65–81. Arlington, VA, USA. http://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf.

Hevner, Alan R., Salvatore T. March, Jinsoo Park ja Sudha Ram. 2004. “Design science in information systems research”. *MIS Quarterly* (Minneapolis, MN, USA) 28 (1): 75–105. https://dl.acm.org/citation.cfm?id=2017217.

Hickson, Ian, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O’Connor ja Silvia Pfeiffer. 2014. *HTML5.* Verkossa; viitattu 17. marraskuuta 2015. http://www.w3.org/TR/2014/PR-html5-20140916/.

Jazayeri, M. 2007. “Some Trends in Web Application Development”. Teoksessa *Future of Software Engineering, 2007. FOSE ’07,* 199–213. Toukokuu. doi:10.1109/FOSE.2007.26.

Kerschbaum, Florian. 2007. “Simple cross-site attack prevention”. Teoksessa *3rd International Conference on Security and Privacy in Communication Networks,* 464–472. doi:10.1109/SECCOM.2007.4550368.

Komanduri, Saranga, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor ja Serge Egelman. 2011. “Of Passwords and People: Measuring the Effect of Password-composition Policies”. Teoksessa *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,* 2595–2604. CHI ’11. Vancouver, BC, Canada: ACM. doi:10.1145/1978942.1979321.

Krishnamurthy, B., ja C.E. Wills. 2009. “On the leakage of personally identifiable information via online social networks”. Teoksessa *Proceedings of the 2nd ACM workshop on Online social networks,* 7–12. ACM. doi:10.1145/1592665.1592668.

Laskos, Tasos. 2015a. *Crawl coverage and vulnerability detection - Arachni - Web Application Security Scanner Framework.* Verkossa; viitattu 17. marraskuuta 2015. http://www.arachni-scanner.com/features/framework/crawl-coverage-vulnerability-detection/.

———. 2015b. *Home - Arachni - Web Application Security Scanner Framework.* Verkossa; viitattu 17. marraskuuta 2015. http://www.arachni-scanner.com/.

Li, Xiaowei, ja Yuan Xue. 2011. *A Survey on Web Application Security.* Tekninen raportti. Vanderbilt University. http://www.truststc.org/pubs/814.html.

MacFarlane, John. 2015. *Pandoc - Pandoc User’s Guide.* Verkossa; viitattu 17. marraskuuta 2015. http://pandoc.org/README.html.

Magnabosco, John. 2009. *Protecting SQL Server Data.* Toimittanut Tony Davis. Red Gate Software.

March, Salvatore T., ja Gerald F. Smith. 1995. “Design and natural science research on information technology”. *Decision Support Systems* 15 (4): 251–266. doi:10.1016/0167-9236(94)00041-2.

Muñoz, F Román, ja LJ García Villalba. 2013. “Methods to Test Web Application Scanners”. Teoksessa *Proceedings of the 6th International Conference on Information Technology.* http://sce.zuj.edu.jo/icit13/images/Camera%20Ready/Computers%20and%20Networks%20Security/767.pdf.

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

Provos, Niels, ja David Mazières. 1999. "A Future-Adaptable Password Scheme". Teoksessa *Proceedings of the Annual Conference on USENIX Annual Technical Conference,* 32–32. ATEC '99. Monterey, California: USENIX Association. http://dl.acm.org/citation.cfm?id=1268708.1268740.

Riancho, Andres. 2015. *Advanced use cases - w3af - Web application attack and audit framework 1.7.6 documentation.* Verkossa; viitattu 17. marraskuuta 2015. http://docs.w3af.org/en/latest/advanced-use-cases.html.

Seshadri, Shyam, ja Brad Green. 2014. *AngularJS: Up and Running: Enhanced Productivity with Structured Web Apps.* O'Reilly Media, Inc.

Shar, L.K., ja Hee Beng Kuan Tan. 2013. "Defeating SQL Injection". *Computer* 46, numero 3 (maaliskuu): 69–77. doi:10.1109/MC.2012.283.

Shar, Lwin Khin, ja Hee Beng Kuan Tan. 2012. "Defending against Cross-Site Scripting Attacks". *Computer* 45 (3): 55–62. doi:10.1109/MC.2011.261.

Shirey, Robert W. 2007. *Internet Security Glossary, Version 2.* Verkossa; viitattu 17. marraskuuta 2015. https://tools.ietf.org/html/rfc4949.

Stuttard, Dafydd, ja Marcus Pinto. 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws.* Wiley.

Taneski, V., M. Heričko ja B. Brumen. 2014. "Password security - No change in 35 years?" Teoksessa *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on,* 1360–1365. Toukokuu. doi:10.1109/MIPRO.2014.6859779.

Turnbull, James. 2014. *The Docker Book: Containerization is the new virtualization.* James Turnbull.

Ur, Blase, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro ym. 2012. “How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation”. Teoksessa *Proceedings of the 21st USENIX Conference on Security Symposium,* 65–80. Security’12. Bellevue, WA: USENIX Association. http://dl.acm.org/citation.cfm?id=2362793.2362798.

Vacca, John R. 2013. *Computer and Information Security Handbook, Second Edition.* 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Weinberger, Joel, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin ja Dawn Song. 2011. “A Systematic Analysis of XSS Sanitization in Web Application Frameworks”. Teoksessa *Proceedings of the 16th European Conference on Research in Computer Security,* 150–171. ESORICS’11. Leuven, Belgium: Springer-Verlag. doi:10.1007/978-3-642-23822-2_9.

Williams, J, ja D Wichers. 2013. *OWASP Top Ten – 2013.* Verkossa; viitattu 17. marraskuuta 2015. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf.

Visaggio, C.A, ja L.C. Blasio. 2010. “Session management vulnerabilities in today’s web”. *Security Privacy, IEEE* 8, numero 5 (syyskuu): 48–56. doi:10.1109/MSP.2010.114.

Wood, Paul, Ben Nahorney, Kavitha Chandrasekar, Scott Wallace ja Kevin Haley. 2014. *Internet Security Threat Report 2014.* Tekninen raportti Volume 19. Symantec Corporation, huhtikuu. https://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf.

Zalewski, Michal. 2012. *The Tangled Web: A Guide to Securing Modern Web Applications.* No Starch Press.

Zeller, W., ja E.W. Felten. 2008. *Cross-site request forgeries: Exploitation and prevention.* Tekninen raportti. Princeton University. https://www.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf.

# Liitteet

## A SQL-kyselyt, joissa `execute`-metodin ensimmäinen parametri ei ole vakiomerkkijono

```
@contract
def getUsersForTasks(self, task_ids: 'list(str)') -> 'list(dict)':
    cursor = self.db.cursor()
    placeholder = '?'
    placeholders = ', '.join(placeholder for unused in task_ids)
    cursor.execute(
        """
            SELECT User.id, name, real_name,
                   COUNT(DISTINCT task_id) AS task_count FROM User
            JOIN UserAnswer ON User.id = UserAnswer.user_id
            JOIN Answer ON Answer.id = UserAnswer.answer_id
            WHERE task_id IN (%s)
            GROUP BY User.id
            ORDER BY real_name ASC
        """ % placeholders, task_ids)

    return self.resultAsDictionary(cursor)

@contract
def getAnswersForGroup(self, user_ids: 'list(int)', task_id: 'str')
                       -> 'list(dict)':
    cursor = self.db.cursor()
    sql = """select id, task_id, content, points, answered_on
             from Answer where task_id = ?
                      %s
             order by answered_on desc""" % (
    " ".join(["""and id in (select answer_id from UserAnswer
              where user_id = %d)""" % user_id for user_id in user_ids]))
    cursor.execute(sql, [task_id])
    return self.resultAsDictionary(cursor)

@contract
def getDocumentsByIds(self, document_ids: 'list(int)') -> 'seq(row)':
    cursor = self.db.cursor()
    cursor.execute("""select id,description as name
                      from Block where id in (%s)""" %
                   ','.join('?' * len(document_ids)), document_ids)
    return cursor.fetchall()

def getParMappings(self, doc_id : 'DocIdentifier',
                   start_index = 0, end_index = -1) -> 'list(tuple)':
    cursor = self.db.cursor()
    endclause = " and par_index < {}"\
        .format(end_index) if end_index >= 0 else ""
```

```
    cursor.execute(
        """select par_index, new_index
           from ParMappings
           where doc_id = ?
             and doc_ver = ?
             and par_index >= ?""" + endclause,
        [doc_id.id, doc_id.hash, start_index])
    return cursor.fetchall()

def clear(self):
    for table in TABLE_NAMES:
        self.db.execute('delete from ' + table)

@contract
def getMappedValues(self, UserGroup_id: 'int|None', doc_id: 'int',
                    doc_ver: 'str', table: 'str',
                    status_unmodified="unmodified",
                    status_modified="modified", extra_fields=None,
                    custom_access: 'str'='0', order_by_sql: 'str'='')
                    -> 'list(dict)':
    if not extra_fields:
        extra_fields = []
    cursor = self.db.cursor()
    fields = ['par_index', 'doc_ver', 'deprecated'] + extra_fields

    query = "select {} from {} where ({}) and doc_id = ? {}"\
        .format(','.join(fields),
                table,
                'UserGroup_id = ? OR ({})'.format(custom_access)\
                if UserGroup_id is not None else custom_access,
                order_by_sql)

    cursor.execute(query, [UserGroup_id, doc_id]\
                   if UserGroup_id is not None else [doc_id])
    # ...
    cursor.execute("""update {} set par_index = ?,
                                    doc_ver = ?,
                                    deprecated = ?
                      where par_index = ?
                      and doc_ver = ? and doc_id = ?""".format(table), [
                   par_index_new,
                   doc_ver,
                   modified,
                   par_index,
                   read_ver,
                   doc_id
                   ])
    # ...
    cursor.execute("""delete from {}
                      where par_index = ?
                            and doc_ver = ?
                            and doc_id = ?""".format(table), [
                   par_index,
```

```
                    read_ver,
                    doc_id
                    ])
```

# B Arachni-työkalun skannausprofiilin asetukset YAML-muodossa

```
---
audit:
  parameter_values: true
  exclude_vector_patterns: []
  include_vector_patterns: []
  link_templates: []
  links: true
  forms: true
  cookies: true
  headers: true
  with_both_http_methods: true
  cookies_extensively: false
datastore: {}
session:
  check_url: http://192.168.59.103:50000/checkLogin
  check_pattern: "(?-mix:1)"
http:
  user_agent: Arachni/v1.0.6
  request_timeout: 50000
  request_redirect_limit: 5
  request_concurrency: 1
  request_queue_size: 500
  request_headers: {}
  cookies: {}
browser_cluster:
  pool_size: 6
  job_timeout: 120
  worker_time_to_live: 100
  ignore_images: false
  screen_width: 1600
  screen_height: 1200
input:
  values:
    "(?i-mx:name)": arachni_name
    "(?i-mx:user)": arachni_user
    "(?i-mx:usr)": arachni_user
    "(?i-mx:pass)": 5543!%arachni_secret
    "(?i-mx:txt)": arachni_text
    "(?i-mx:num)": '132'
    "(?i-mx:amount)": '100'
    "(?i-mx:mail)": arachni@email.gr
    "(?i-mx:account)": '12'
    "(?i-mx:id)": '1'
  without_defaults: true
```

```
  force: false
scope:
  redundant_path_patterns: {}
  dom_depth_limit: 20
  exclude_path_patterns: []
  exclude_content_patterns: []
  include_path_patterns: []
  restrict_paths: []
  extend_paths:
  - "/manage/1"
  - "/view/1"
  - "/teacher/Dokumentti%201"
  url_rewrites: {}
  include_subdomains: false
  https_only: false
checks:
- code_injection
- code_injection_php_input_wrapper
- code_injection_timing
- csrf
- file_inclusion
- ldap_injection
- no_sql_injection
- no_sql_injection_differential
- os_cmd_injection
- os_cmd_injection_timing
- path_traversal
- response_splitting
- rfi
- session_fixation
- source_code_disclosure
- sql_injection
- sql_injection_differential
- sql_injection_timing
- trainer
- unvalidated_redirect
- unvalidated_redirect_dom
- xpath_injection
- xss
- xss_dom
- xss_dom_inputs
- xss_dom_script_context
- xss_event
- xss_path
- xss_script_context
- xss_tag
- xxe
- allowed_methods
- backdoors
- backup_directories
- backup_files
- captcha
- common_directories
```

```
- common_files
- cookie_set_for_parent_domain
- credit_card
- cvs_svn_users
- directory_listing
- emails
- form_upload
- hsts
- htaccess_limit
- html_objects
- http_only_cookies
- http_put
- insecure_client_access_policy
- insecure_cookies
- insecure_cors_policy
- insecure_cross_domain_policy_access
- insecure_cross_domain_policy_headers
- interesting_responses
- localstart_asp
- mixed_resource
- origin_spoof_access_restriction_bypass
- password_autocomplete
- private_ip
- ssn
- unencrypted_password_forms
- webdav
- x_frame_options
- xst
platforms:
- linux
- sqlite
- python
plugins:
  autologin:
    url: http://192.168.59.103:50000/login?emailLogin=1
    parameters: email=arachni@test.com&password=password
    check: successfully logged in
  autothrottle:
  discovery:
  healthmap:
  timing_attacks:
  uniformity:
no_fingerprinting: false
authorized_by:
name: TIM-scan
description: TIM-scan
```

# C TIM-sovelluksen Dockerfile

```
# Start from the Ubuntu image
```

```
2  from ubuntu
3  maintainer Ville Tirronen "ville.tirronen@jyu.fi"
4
5  # Set locale
6  env LANG en_US.utf8
7  env LANGUAGE en_US.utf8
8  run locale-gen en_US.utf8
9
10 # Install Python, pip and other necessary packages
11
12 run apt-get update
13 run apt-get install -y python3
14 run apt-get install -y python3-pip
15 run apt-get install -y git-core
16
17 run apt-get install -y zlib1g-dev # lxml dependency
18 run apt-get install -y libxml2-dev libxslt-dev python3-dev # lxml
       dependency
19 run apt-get install -y libyaml-dev # C-parser for PyYAML
20
21 run pip3 install flask
22 run pip3 install flask-compress
23 run pip3 install beautifulsoup4
24 run pip3 install pycontracts
25 run pip3 install hypothesis
26 run pip3 install gitpylib
27 run pip3 install lxml
28 run pip3 install pyaml
29 run pip3 install ansiconv
30 run pip3 install cssutils
31
32 # Install pygit2 and dependencies
33 run apt-get install -y wget
34 run apt-get install -y cmake
35 run wget https://github.com/libgit2/libgit2/archive/v0.22.0.tar.gz && tar
       xzf v0.22.0.tar.gz
36 run cd libgit2-0.22.0/ && cmake -DPYTHON_EXECUTABLE=/usr/bin/python3 . &&
       make install
37
38 run apt-get install -y libffi-dev
39 env LD_LIBRARY_PATH /usr/local/lib
40 run pip3 install pygit2
41
42 # Need to update the package "six"; otherwise there will be a strange
       error with Flask's reloader
43 # run pip3 install -U six
44
45 # requests must be the last package to be installed!
46 # After this, pip3 stops working because it depends
47 # on older version of requests package.
48 run pip3 install requests --upgrade
49
```

```
50 # Remove pip. It's no longer needed, we get slightly smaller image and
       less programs
51 # for a potential attacker to use.
52 # UPDATE: Don't remove pip; it causes problems with running tests.
53 # run apt-get -y --purge remove python3-pip
54
55 # Set name and email for git.
56 run git config --global user.email "agent@docker.com"
57 run git config --global user.name "agent"
58
59 run mkdir /service
60
61 # Add user `agent` -- we don't want to run anything as root.
62 run useradd -M agent
63 run chown -R agent /service
64
65 expose 5000
66 expose 22
67
68 run apt-get install -y npm
69 run npm install -g bower
70 run ln -s /usr/bin/nodejs /usr/bin/node
71 run bower --allow-root install angular#~1.3
72 run bower --allow-root install angular-ui-ace#36844
       ff7c0e0d9445bc8e31514d7f0f59cb8b048
73 run bower --allow-root install ng-file-upload#~3.0
74 run bower --allow-root install jquery#~2.1
75 run bower --allow-root install angular-sanitize#~1.3
76 run bower --allow-root install bootstrap#~3.3
77 run bower --allow-root install waypoints#~3.1
78
79 # We need to patch the ui-ace module to fix this issue: https://github.
       com/angular-ui/ui-ace/issues/27
80 run sed -i 's/var opts = angular\.extend({}, options, scope\.$eval(attrs
       \.uiAce));/var opts = angular.extend({}, options, scope.$eval(attrs.
       uiAce, scope));/' /bower_components/angular-ui-ace/ui-ace.js
81
82 run npm uninstall bower
83 run apt-get remove -y npm
84
85 run apt-get install -y openssh-server
86 run mkdir /var/run/sshd
87 run echo 'root:test' | chpasswd
88 run sed -i 's/PermitRootLogin without-password/PermitRootLogin yes/' /etc
       /ssh/sshd_config
89
90 run apt-get autoremove -y
91
92 # Configure timezone and locale
93 run echo "Europe/Helsinki" > /etc/timezone; dpkg-reconfigure -f
       noninteractive tzdata
94
95 # Default startup command
```

```
96 cmd cd /service/timApp && source initenv.sh && python3 launch.py
```