Ohjelmistojen tietoturvan parantaminen haavoittuvuuksia poistamalla

Pasi Hiltunen

6.4.2005

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Tutkielmassa tarkastellaan yleisimpiä ohjelmistojen tietoturvauhkia sekä tapoja, joiden avulla ohjelmistoissa olevia haavoittuvuuksia on mahdollista lieventää tai poistaa. Tutkielman alussa käsitellään ohjelmiston koko elinkaarta tietoturvallisuuden kannalta, mutta loppuosa keskittyy pelkästään ohjelmiston suunnittelu- ja kehitysvaiheissa tehtäviin tietoturvallisuutta edistäviin ratkaisuihin. Tutkielman pääaihe on uhkamallinnus, joka on riskienhallintapohjainen lähestymistapa ohjelmistojen tietoturvaan liittyvien haavoittuvuuksien poistamiseksi. Tutkielmassa käsitellään myös hyökkäysmalleja, joita voidaan käyttää uhkamallinnuksen aikana suoritettavan uhkien tunnistamisen hyödyntämien hyökkäyspuiden apuna. Tutkielman lopussa käsitellään tarkastuslistoja sekä joitakin yleisimpiä tietoturvaperiaatteita, joiden käyttö osaltaan lisää ohjelmistojen turvallisuutta.

ACM-luokitus (ACM Computing Classification System, 1998 version): D.4.6, K.6.5

Avainsanat: haavoittuvuus, hyökkäys, hyökkäysmalli, hyökkäyspuu, tietoturva, uhka, uhkamallinnus

Sisältö

1. Johdanto 1
    1.1 Tietoturvavahingot 1
    1.2 Ohjelmiston tietoturvan parantaminen 2
    1.3 Tutkielman rakenne 5
2. Hyökkäysmallit 8
    2.1 Muuttujan injektointi 9
    2.2 Skriptien upottaminen skriptittömiin elementteihin 11
    2.3 Vaihtoehtoiset IP-osoitteet 12
    2.4 Muuttujien ja tunnisteiden ylivuoto 14
3. Turvallisuuden suunnittelu uhkamallinnuksen avulla 17
    3.1 Ohjelmiston jakaminen osiin 18
    3.2 Järjestelmän uhkien määrittely 24
        3.2.1 STRIDE-mallin käyttö uhkien ryhmittelyyn 24
        3.2.2 Hyökkäyspuun käyttö järjestelmän uhkien määrittelyssä 26
        3.2.3 Uhkamallinnuksen aikana löydetyistä uhkista kirjattavat asiat 29
    3.3 Uhkien luokittelu kohoavan riskin mukaan 31
    3.4 Tapa vastata kuhunkin uhkaan 34
    3.5 Tekniikat joiden avulla uhkia voidaan lieventää 35
4. Periaatteet ja tarkastuslistat 39
    4.1. Periaatteita ja käytäntöjä tietoturvan parantamiseksi 39
        4.1.1 Heikoimman lenkin turvaaminen 39
        4.1.2 Perusteellisten suojauksien käyttäminen 40
        4.1.3 Epäonnistuminen turvallisesti 40
        4.1.4 Pienimmän mahdollisen oikeuden periaatteen käyttäminen 41
        4.1.5 Osiin jakaminen 41
        4.1.6 Hyökkäyskohteiden vähentäminen 42
        4.1.7 Yksityisyyden edistäminen 42
        4.1.8 Salaisuuksien piilottamisen vaikeus 43
        4.1.9 Luottamuksen rajoittaminen 45
        4.1.10 Yhteisön resurssien hyödyntäminen 46

4.1.11 Tuotteen turvallisuustavoitteiden määrittely 46
4.1.12 Virheistä oppiminen 47
4.2. Tarkastuslistat 48
5. Yhteenveto 52
Viitteet 54
LIITE 1: Kirjautumistunnusten tarkastukseen käytettävä login.asp-tiedosto 56

1. Johdanto

Tietoturva voi tarkoittaa eri henkilöille eri asioita, mutta yleensä tietoturvalla tarkoitetaan hallinnollisia ja teknisiä toimenpiteitä, joiden avulla taataan tiedon luottamuksellisuus, eheys ja käytettävyys. Ohjelmistojen tietoturva riippuu monesta eri tekijästä, kuten muun muassa siitä miten ohjelmisto kestää sitä vastaan tehtyjä hyökkäyksiä ja miten vähän ohjelmassa on virheitä. Turvallisuuden tulisi olla osa tuotetta ja tuotteen suunnitelussa alussa asti mukana oleva turvallisuussuunnitelu varmistaa myös sen, että tuote ja turvallisuus toimivat yhdessä. Jälkikäteen lisätty turvallisuus lisää turvallisuuteen liittyvien virheiden määrää verrattuna alusta asti mukana olevaan turvallisuuteen. Ohjelmiston jokaisen tietoturvan osa-alueen tulisi olla hyvin suunniteltu ja toteutettu, että ohjelmisto olisi luotettava.

1.1 Tietoturvavahingot

McGrawn (2002) artikkelin mukaan ohjelmistojen tietoturvavahinkojen lisääntyminen johtuu pääasiallisesti kolmesta seikasta. Ensimmäisenä seikkana on *verkkoyhteyksien* määrän jatkuva *lisääntyminen*, minkä takia hyökääjä ei tarvitse fyysistä yhteyttä hyökätäkseen järjestelmään, vaan hyökkäys voidaan toteuttaa verkon välityksellä. Toinen merkittävä asia on *laajennettavuus*. Monet ohjelmistot tukevat dynaamisesti ladattavia ajureita ja moduuleja, mikä lisää ohjelmistojen haavoittuvuutta, koska on mahdollista päivittää ohjelmisto myös hyökkääjän muokkaamilla laajennuksilla. Viimeinen tietoturvavahinkojen lisääntymistä aiheuttava seikka on *monimutkaisuus*, joka lisää ohjelmistojen kokoa ja samalla mahdollisia hyökkäyskohteita.

Lisääntyvässä määrin hyökkäyksen motiivina on hyökkääjän taloudellinen hyötyminen (Viestintävirasto, 2004), mutta hyökkäys voi olla myös kiusantekoa tai vain kokeilua, voidaanko järjestelmään murtautua. Hyökkäykset ovat viime aikoina myös ammattimaistuneet, mikä on osaltaan vaikuttanut hyökkäysten onnistumiseen ja hyökkäyksen aiheuttamiin vahinkoihin. Nikander & al. (1996) jakavat tietoturvavahingot *potentiaalisiin vahinkoihin, tietoihin kohdistuviin vahinkoihin* ja *laitteistoon/ohjelmistoon kohdistuviin vahinkoihin*.

Potentiaalisilla vahingoilla tarkoitetaan järjestelmän varsinaiselle toiminnalle harmittomia, mutta toiminnan vaarantavia tai selkeästi rikollisia tapahtumia, kuten esimerkiksi varmuuskopioinnin laiminlyönti, virheellisten tai mahdollisesti viruksia sisältävien ohjelmien käyttö ja levitys sekä ulkopuolisen tunkeutuminen järjestelmään, joka voidaan toteuttaa tietoliikenteen tai hallintapäätteen avulla. Ulkopuolisen tekemä järjestelmän resurssien tuhlaaminen on myös potentiaalinen vahinko, joka voi tapahtua monin eri tavoin, tuhlattavana olevasta resurssista riippuen. On mahdollista kuormittaa laitteiston eri resursseja niin, että siitä on haittaa muille käyttäjille. Tietoliikennettä on mahdollista käyttää resurssien tuhlaamiseen myös järjestelmän ulkopuolelta. Esimerkiksi järjestelmälle voidaan aiheuttaa niin sanottu *palvelunestohyökkäys* lähettämällä järjestelmälle jatkuvasti sähköposti- tai muuta tietoliikennettä.

Tietoon kohdistuvia vahinkoja ovat muun muassa tietojen vuoto eli tietojen valtuuttamaton käyttö, joka voi tapahtua joko järjestelmän sisällä silloin, kun käyttäjä pääsee käsiksi hänelle valtuuttamattomaan tietoon tai salakuuntelemalla järjestelmän ulkopuolelta. Toinen tietoon kohdistuva vahinko on tietojen tuhoutuminen tai vahingoittuminen, joka voi tapahtua pahantahtoisen käyttäjän tai tunkeutujan toimesta, mutta todennäköisempi syy lienee laitteistoon tai ohjelmistoon tullut virhe tai käyttäjän tekemä virhe, jonka seurauksena tiedot katoavat.

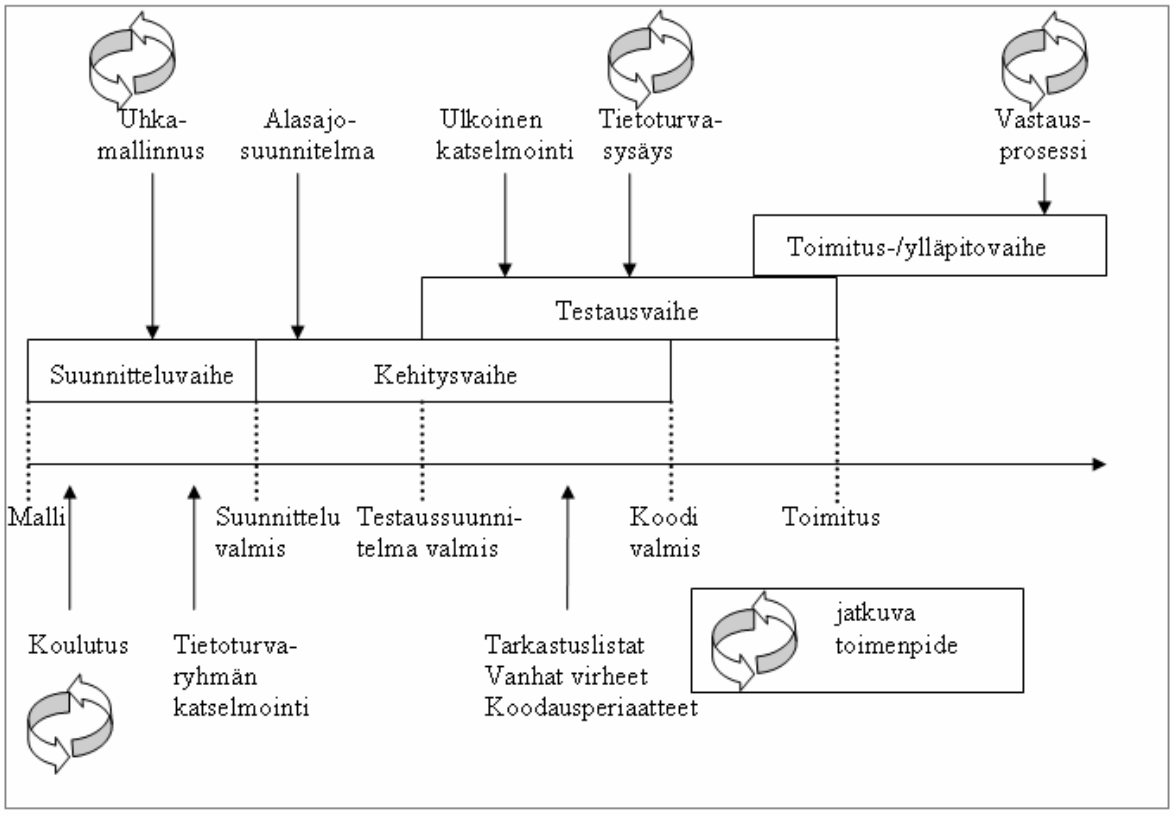
Laitteistoon tai ohjelmistoon kohdistuvia vahinkoja ovat muun muassa ohjelmistojen muuttaminen, joka voi olla tehty pelkästään ilkivaltatarkoituksessa tai muutoksen tarkoituksena voi olla uuden tietoturva-aukon avaaminen. Laitteistoja voidaan myös tuhota tai varastaa. Laitteistoa ja ohjelmia koskevia tahattomia virheitä synnyttää osaltaan alan nopea kehitys. Tuottaakseen ajoissa markkinoille kysyntää parhaiten vastaavan tuotteen laite- tai ohjelmistovalmistajalla on yleensä todella kiire. On vaara, että tuote valmistuu markkinoille jo valmiiksi vanhanaikaisena.

1.2 Ohjelmiston tietoturvan parantaminen

Ohjelmistojen tietoturvan tulisi olla osa koko ohjelmiston elinkaarta, eikä vain yksi toteutuksen loppuvaiheessa lisätty ominaisuus (McGraw ja Viega, 2002). Ohjelmistojen tietoturvan

päämäärät ovat usein ristiriidassa joidenkin ohjelmistoprojektin päämäärien kanssa. Ohjelmistoprojektin tärkeimpiä päämääriä ovat yleensä t*oiminnallisuus*, sekä *käytettävyys,* jota tietoturvamekanismit kuten salaus ja kirjautumismenttelyt usein heikentävät. Päämääristä *tehokkuus* jää usein turvallisuuden jalkoihin, kuten esimerkiksi etäpalvelimen suorittaman tunnistuksen muodossa. Ohjelmistoprojekteja ja turvallisuutta koskeva *yksinkertaisuus* on hyvä asia molemmille, mutta *markkinoille tulon nopeus* on usein ristiriidassa turvallisuuden kanssa, koska tingittäessä ohjelmiston tuottamiseen käytetystä ajasta, tingitään yleensä ohjelmiston riskienhallintaan käytetystä ajasta.

Ohjelmistojen turvallisuuden parantaminen edellyttää tietoturvan huomioon ottamista ohjelmistoprosessin jokaisessa vaiheessa (Howard ja LeBlanc, 2003). Prosessiin on mahdollista lisätä parannuksia jokaisessa ohjelmistojen kehityselinkaaren vaiheeessa, riippumatta siitä, mitä elinkaarimallia käytetään.



Kuva 1: Turvallisuuden liittyminen ohjelmiston elinkaareen (Howard ja LeBlanc, 2003).

Howard ja LeBlanc (2003) jakavat kuvan 1 mukaisesti ohjelmiston elinkaaren karkeasti neljään eri vaiheeseen, joista ensimmäisenä on *suunnitteluvaihe,* jolloin myös tietoturvasuunnitelmat on tärkeää saada kuntoon. Ohjelmointivirheen korjaaminen kehitys- tai testausvaiheessa vaatii paljon enemmän aikaa, rahaa ja työtä kuin virheen korjaaminen jo suunnitteluvaiheen aikana. On myös helpompaa korjata jotain, jos sitä ei tarvitse korjata sen takia, että se oli väärin suunniteltu. Näiden seikkojen takia on tärkeää saada tietoturvatavoitteet ja suunnittelu kuntoon niin aikaisessa vaiheessa kuin mahdollista.

Suunnitteluvaiheesta lähtien, tulisi tarvittaessa järjestää myös koulutusta. Koulutuksen tulisi keskittyä siihen, kuinka tietoturvaa lisääviä ominaisuuksia voidaan hyödyntää, pääpainon ei siis tulisi olla tietoturvaa lisäävien ominaisuuksien teknisissä ja teoreettisissa ratkaisuissa. Koulutusta tulisi myös järjestää riittävän usein, koska turvallisuuden kuva muuttuu jatkuvasti, uusien uhkien ilmestyessä.

Seuraava ja samalla tärkein asia tietoturvan osalta suunnitteluvaiheessa on aloittaa *uhkamallinnus*, osana riskienhallintaa. Riskienhallinnan tarkoituksena on pitää riskit hyväksyttävällä tasolla. Uhkamallinnuksen päävaiheet ovat uhkien tunnistaminen ja niiden suuruuden arviointi, uhkien merkityksen arviointi sekä uhkien valvonta, joka sisältää toimenpiteet muun muassa uhkien lieventämiseksi. Ilman uhkamallinnusta ei voida tehdä turvallisia järjestelmiä, koska turvallisen järjestelmän luominen vaatii, että ymmärretään järjestelmän uhat. Suunnitteluvaiheessa tulisi luoda myös *alasajaosuunnitelmat* vanhalle toiminnallisuudelle päätettäessä luopua vanhasta ominaisuudesta ja korvata se uudella ja turvallisemmalla ominaisuudella.

Seuraava ohjelmiston elinkaaren vaihe on *kehitysvaihe,* joka sisältää ohjelmakoodin kirjoittamista ja virheiden korjaamista. Kehitysvaiheessa tulisi keskittyä siihen, että ohjelmiston kehittäjät kirjoittavat niin hyvälaatuista koodia kuin mahdollista. Tätä voidaan edesauttaa määrittelemällä *tietoturvallisen koodauksen periaatteet* ja *tarkastuslistat*. Kehitysvaiheessa tulisi myös tarkastella *vanhoja virheitä* ja niistä tulisi oppia niin, että samoja virheitä ei enää toistettaisi.

Koulutus yhdistettynä tietoturvallisuuteen painottuneen *koodikatselmoinnin* kanssa lisää koodin turvallisuutta. Turvallisuus ei parane pelkästään sen takia, että koodin laatu tarkastetaan tietoturvallisuuden kannalta, vaan myös sen takia, että kehittäjä tietää, että toiset henkilöt tulevat tarkastelemaan hänen koodiaan. Tätä kutsutaan Hawthornin vaikutukseksi. Mahdollisuuksien salliessa on kannattava hyödyntää myös jotakin *ulkopuolista tahoa*, joka suorittaa koodille ja suunnitelmille oman katselmoinnin. Tällä tavoin saadaan tietoturva-asioihin myös ulkopuolinen näkemys.

Viimeistään kehitysvaiheessa tulisi aloittaa *tietoturvasysäykset* (security push), joiden tavoitteena on parantaa tiimin jäsenten tietoturvatietoisuutta, sekä löytää ja korjata ohjelmakoodissa olevia tietoturvapuutteita.

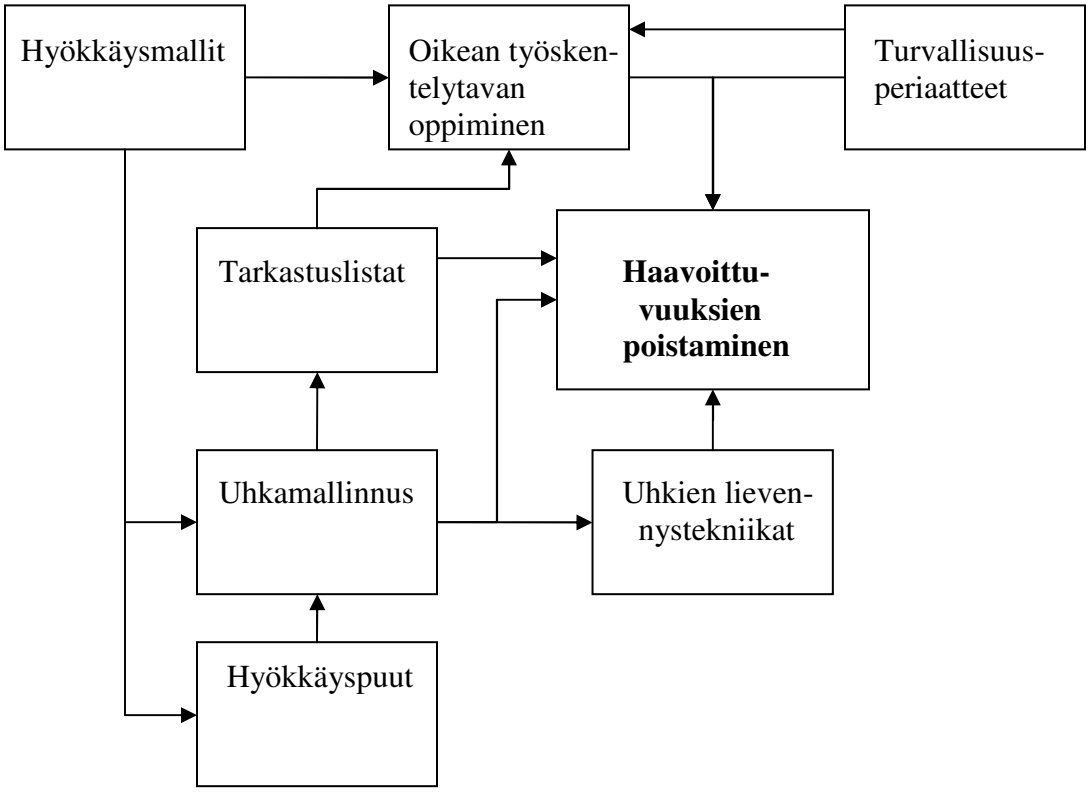
Ohjelmistojen elinkaaren *testausvaihe* on tärkeä vaihe ohjelmistokehityksessä. Testaajat tulee kouluttaa niin, että he tietävät kuinka hyökkääjät toimivat ja lisäksi heidän täytyy oppia samat tietoturvatekniikat kuin ohjelmoijien. McGrawn (2004) mukaan testauksen tulisi sisältää *riskipohjainen tietoturvatestaus*, joka perustuu hyökkäysmalleihin ja uhkamallinnukseen.

Ohjelmiston elinkaaren viimeisinä vaiheina ovat *toimitus- ja ylläpitovaiheet*. Howardin ja LeBlancin (2003) mukaan tuotetta ei saisi toimittaa, jos siinä on tunnettuja hyödynnettävissä olevia haavoittuvuuksia. Jos tuotteesta löytyy toimituksen jälkeen tietoturvaongelmia, tulisi niihin vastata jollain tavalla. Jos tuote päädytään korjaamaan, niin haavoittuvan ohjelmakoodin kirjoittaneen henkilön tulisi suorittaa myös siinä olevan virheen korjaaminen. Tällä tavoin ohjelmoijalla olisi mahdollisuus oppia virheistään.

1.3 Tutkielman rakenne

Tässä tutkielmassa tutkitaan tapoja ohjelmistojen haavoittuvuuksien lieventämiseksi ja poistamiseksi kuvan 2 esittämällä tavalla. *Haavoittuvuudella* (vulnerability) tarkoitetaan ohjelmistossa olevaa suunnittelu- tai toteutusvirhettä, jonka seurauksena ohjelmistoa vastaan on mahdollista hyökätä. Yksi yleisimmistä ohjelmistoissa olevista haavoittuvuuksista on ylivuodon aiheuttama haavoittuvuus. Ohjelmistojen tietoturva on tärkeässä osassa ohjelmistojen

käytettävyyttä ajateltaessa, koska ohjelmistojen haavoittuvuuksien hyödyntäminen lisääntyy jatkuvasti. Tutkielma keskittyy ohjelmistojen turvallisuuden parantamiseen ohjelmistojen suunnittelu- ja kehitysvaiheissa. Tietoturvallisuuden kannalta tehokkaasti suunnitellulla ja toteutetulla ohjelmistolla on mahdollisuuksia suojautua jo tiedossa olevilta ja myös tulevaisuudessa kehitettäviltä uusilta uhkilta.



Kuva 2: Tutkielman rakenne.

Ennen kuin järjestelmän haavoittuvuuksien poistaminen olisi mahdollista, tulee ensin tunnistaa järjestelmän uhat kohdan 3.2 mukaisesti osana uhkamallinnusta. Uhkien tunnistamiseksi tutkielmassa käytetään alakohdan 3.2.2 *hyökkäyspuita*, joiden avulla voidaan kuvata hyökkääjien uhan hyödyntämiseksi käyttämää päätöksentekoprosessia. Hyökkäyspuussa puun juuri kuvaa hyökkääjien päämäärää ja lehdet kuvaavat tapoja joilla haluttu päämäärä voidaan saavuttaa. Hyökkäyspuissa tarvittavien päämäärien ja toimintatapojen hankkimiseen tutkielman luvussa 2 esitellään *hyökkäysmalleja*, jotka ovat kuvauksia siitä, kuinka ohjelmiston haavoittuvuutta voidaan käyttää hyväksi.

Luvussa 3 esiteltävän *uhkamallinnuksen* aikana uhkia voidaan tunnistaa myös käyttämällä pelkkiä hyökkäysmalleja, mutta tällöin ei välttämättä tiedetä, voiko hyökkäysmallin esittämä uhka muodostua haavoittuvuudeksi tutkittavassa järjestelmässä. Uhkamallinnus on eräs riskienhallinnan muoto, jossa tarkoituksena on tunnistaa ja luokitella uhat sekä valita tapa vastata kuhunkin uhkaan. Uhkamallinnuksen muut päävaiheet uhkien tunnistamisen lisäksi ovat kohdassa 3.3 tarkasteltava järjestelmän *uhkien järjestäminen riskin mukaan* sekä kohdassa 3.5 tarkasteltava *lievennystekniikan* valinta, jolla kuhunkin järjestelmän uhkaan vastataan. Uhkamallinnuksessa hankittujen tietojen avulla on mahdollista luoda uusia kuhunkin tilanteeseen parhaiten sopivia uhkien livennystekniikoita.

Haavoittuvuuksien poistamiseksi on mahdollista käyttää myös luvun 4 mukaisesti *turvallisuusperiaatteita* ja *–tarkastuslistoja*. Pelkästään noudattamalla tiettyjä turvallisuusperiaatteita ohjelmistojen suunnittelun ja toteutuksen aikana voidaan suurin osa yleisistä uhkista välttää (McGraw ja Viega, 2002). Tarkastuslistat liittyvät pääasiallisesti kehitysvaiheeseen, jossa tarkastuslistojen avulla on tarkoitus tarkastaa koodia niin, että se täyttää tarkastuslistojen vaatimukset. Tarkastuslistojen laatiminen vaatii uhkamallinnusprosessin läpikäyntiä. Turvallisuusperiaatteet ja tarkastuslistat auttavat osaltaan *oikeiden toimintatapojen oppimisessa*. Uusia turvallisuusperiaatteita on mahdollista luoda aiemmin suoritettujen onnistuneiden uhkamallinnuksen avulla suoritettujen haavoittuvuuksien poistamisen seurauksena.

Luvussa 5 esitetään yhteenveto tutkielmassa käsitellyistä haavoittuvuuksien poistamiseksi käytetyistä seikoista, joiden avulla on mahdollista parantaa ohjelmistojen tietoturvallisuutta.

2. Hyökkäysmallit

Howardin ja LeBlancin (2003) mukaan hyökkäysten tunnistaminen on yksi tärkeimmistä vaiheista ohjelmiston tietoturvan parantamisessa. Jos ei tiedetä hyökkäystapoja, joille järjestelmä on alttiina, ei hyökkäyksiin myöskään ole mahdollista vastata asianmukaisesti.

Hyökkäysmalli on kuvaus siitä, kuinka käyttää hyväksi ohjelmiston haavoittuvuutta. Hoglund ja McGraw (2004) jaottelevat 49 tunnistettua hyökkäysmallia neljään eri ryhmään taulukon 1 mukaisesti hyökkäysmallissa käytetyn tekniikan perusteella.

Taulukko 1: Hyökkäysmallien ryhmittely.

Ryhmä	Lukumäärä
Hyökkäysmallit palvelimen ohjelmiston hyödyntämiseksi	17
Hyökkäysmallit asiakkaan ohjelmiston hyväksikäyttämiseksi	11
Hyökkäysmallit syötteen avulla petkuttamiseksi	10
Hyökkäysmallit puskurien ylivuodon hyödyntämiseksi	11

Palvelimen ohjelmiston hyödyntämiseksi käytetyt hyökkäysmallit keskittyvät pääasiassa Internet-pohjaisiin ohjelmistoihin, mutta myös muut palvelimilla käytetyt ohjelmat saattavat kärsiä samoista haavoittuvuuksista. Palvelimen ohjelmistoa vastaan tehdyt hyökkäykset ovat usein käytettyjä, koska palvelimeen voidaan hyökätä verkon välityksellä. Pääsyy palvelimen ohjelmistojen hyödyntämiseen on siinä, että palvelimet eivät välttämättä suodata käyttäjiltä saatuja syötteitä. Kohdassa 2.1 on tarkasteltu esimerkkinä yhtä palvelimen ohjelmistoa hyödyntävää hyökkäysmallia.

On mahdollista, että hyökkääjä pystyy ottamaan heikosti suojatun palvelimen toiminnan haltuunsa esimerkiksi palvelimen ohjelmiston hyödyntämiseksi tarkoitettujen hyökkäysmallien avulla. Jos hyökkääjä pystyy tunkeutumaan palvelimen kautta asiakkaan ohjelmistoon, niin on mahdollista ladata käyttäjän tiedostoja tai jopa saastuttaa käyttäjän verkko. Kohdassa 2.2 on tarkasteltu esimerkkinä yhtä asiakkaan ohjelmistoa hyväksikäyttävää hyökkäysmallia.

Ohjelmistoilla on yleensä paljon syötteitä, jotka ohjaavat ohjelmiston päätöksentekoa vaikuttamalla ohjelmiston tiloihin. Käyttäjä voi siis syötteiden avulla vaikuttaa suoritettavan ohjelmiston kulloiseenkin tilaan ja viekkaasti suunnitellun syötteen avulla on ohjelmistoa mahdollista käyttää hyväksi saattamalla ohjelmisto haavoittuvaan tilaan. Kohdassa 2.3 on tarkasteltu esimerkkinä yhtä syötteiden avulla petkuttamiseen käytettyä hyökkäysmallia.

Puskurin ylivuoto on todennäköisesti käytetyin tapa ohjelmistojen haavoittuvuuksien hyödyntämisessä Puskurin ylivuotoa hyödyntäviä uusia uhkia paljastuu lähes päivittäin. Etenkin huonosti toteutettua muistinhallintaa käyttävien ohjelmointikielten käyttö tekee puskurin ylivuodon hyödyntämisestä entistä suositumpaa. Kohdassa 2.4 on tarkasteltu esimerkkinä yhtä puskurin ylivuotoa hyödyntävää hyökkäysmallia.

2.1 Muuttujan injektointi

Hoglundin ja McGrawn (2004) mukaan muuttujan injektointiin perustuvaa hyökkäysmallia voidaan hyödyntää silloin, kun käyttäjän syöte annetaan suoritettavalle komennolle ilman suodatusta. Tällaista hyökkäystä voidaan joissakin tapauksissa käyttää esimerkiksi web-sivujen hakukentissä syöttämällä hakusanan sijaan merkkijonon, joka muuttaa koko hakulausekkeen merkityksen. Eräs esimerkki muuttujan injektiosta on SQL-kyselylausekkeen saastutus (Howard ja LeBlanc, 2003).

Tarkastellaan esimerkkinä järjestelmään kirjautumista, jossa tietokannan login-tauluun on tallennettu salaamattomina käyttäjän käyttäjätunnus ja salasana. Tietokannan login-taulun lisääminen Microsoftin SQL Server -tietokantaan on toteutettu käyttämällä SQL Express Manager -ohjelmaa, jolloin login-taulun luontilause on seuraavanlainen:

```
CREATE TABLE login(ssana char(20), ktunnus char(20))
```

Tietojen lisääminen tauluun on suoritettu käyttämällä seuraavanlaista SQL-lausetta:

```
INSERT INTO login VALUES('Pasi', 'sala')
```

Tietokannan login-taulun sisältö SQL-lauseen suorituksen jälkeen on esitetty kuvassa 3.

	ktunnus	ssana
1	Pasi	sala

Kuva 3: Tietokannan sisältö INSERT-lauseen suorituksen jälkeen.

Liitteessä 1 on esitelty login.asp-tiedosto, joka saa syötteenä käyttäjän kirjautumissivulla syöttämän käyttäjätunnuksen ja salasanan. Kirjautumissivulta tiedot välitetään login.asp-tiedostolle käyttäen POST-metodia. Tiedostoa login.asp käytetään tutkittaessa löytyykö tietokannasta käyttäjätunnusta ja salasanaa vastaavaa tietoriviä. Tietorivin tutkimiseen käytetään seuraavanlaista kyselylauseketta, joka palauttaa ehdot toteuttavien tietorivien lukumäärän:

```
SELECT count(*) FROM login
WHERE ktunnus = 'käyttäjätunnus' AND ssana = 'salasana'
```

Jos käyttäjä syöttää käyttäjätunnus- ja salasana-kenttiin esimerkiksi arvot *' OR ''='*, niin muodostuu seuraavankaltainen kysely, joka palauttaa aina yhtä suuremman arvon, koska ''='' on aina tosi:

```
SELECT count(*) FROM login
WHERE ktunnus = '' OR ''='' AND ssana = '' OR ''=''
```

Toisenlainen vaihtoehto on käyttää SQL:n kommenttimerkkejä, kuten esimerkissä käytetyn Microsoftin SQL Serverin tapauksessa kahta väliviivaa (--). Jos hyökkääjä tietää jonkun käyttäjän käyttäjätunnuksen, niin on mahdollista antaa kyselylle käyttäjätunnuksen syötteenä esimerkiksi: *Pasi'* --, joka kommentoi salasanan kyselyn:

```
SELECT count(*) FROM login
WHERE ktunnus = 'Pasi' -- AND ssana = ''
```

Edellä esitetty kysely palauttaa yhtä suuremman arvon, jos login-taulun ktunnus-kentästä löytyy arvo: *Pasi*. Saman kyselyn avulla on myös mahdollista lisätä uusia arvoja esimerkiksi syöttämällä käyttäjätunnus-kenttään arvo: *' INSERT INTO login VALUES ('hacked','pass')* --. Syötteen seurauksena muodostuu seuraavanlainen yhdistetty SQL-lause, joka lisää login-tauluun uudet arvot, joita on mahdollista käyttää kirjautumiseen:

```
SELECT count(*) FROM login
WHERE ktunnus = '' INSERT INTO login VALUES ('hacked','pass') -- AND ssana
= ''
```

Kuva 4 esittää tietokannan sisältöä käyttäjätunnus-kentän avulla suoritetun INSERT-lausekkeen suorituksen jälkeen.

	ktunnus	ssana
1	Pasi	sala
2	hacked	pass

Kuva 4: Tietokannan sisältö käyttäjätunnus-kentän avulla suoritetun injektion jälkeen.

2.2 Skriptien upottaminen skriptittömiin elementteihin

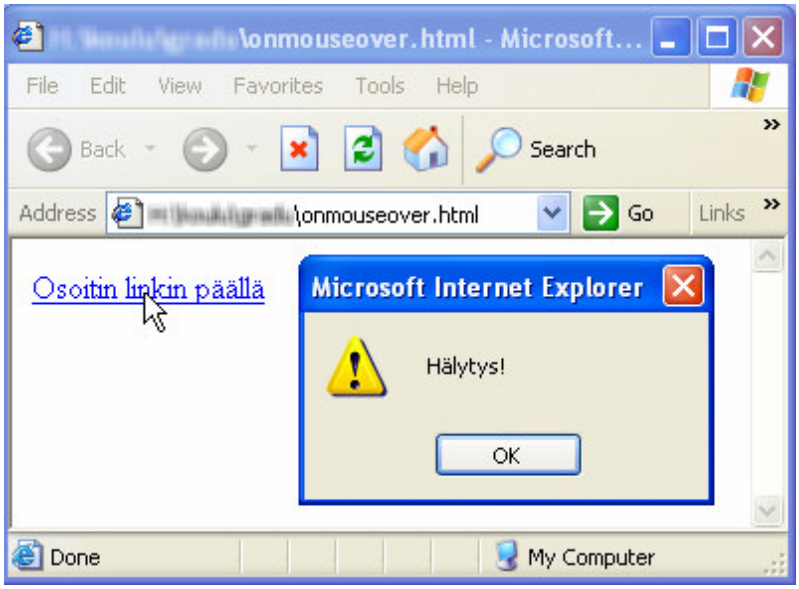
Hyökkäysmallissa tarkastellaan tilanteita, joissa skriptiä ei tarvitse välttämättä lisätä <script>-tunnisteiden väliin. Skripti voi esiintyä toisen HTML-tunnisteen osana, kuten esimerkiksi image-tunnisteena (Hoglund ja McGraw, 2004). Tällöin image-tunniste on muotoa:

```
<img src=javascript:alert(”Hälytys!”)>
```

Skriptejä on mahdollista suorittaa myös pelkästään viemällä hiiren osoitin esimerkiksi linkin tai kuvan päälle, jolloin linkin tai kuvan tunnisteeseen liitetty onMouseover-tapahtuma saa aikaan skriptin suorittamisen:

```
<A HREF="" onMouseover="javascript:alert('Hälytys!')">Osoitin linkin
päällä</A>
```

Kuva 5 esittää onMouseOver-tapahtuman aiheuttamaa hälytystä.



Kuva 5: onMouseover-tapahtuman aiheuttama hälytys.

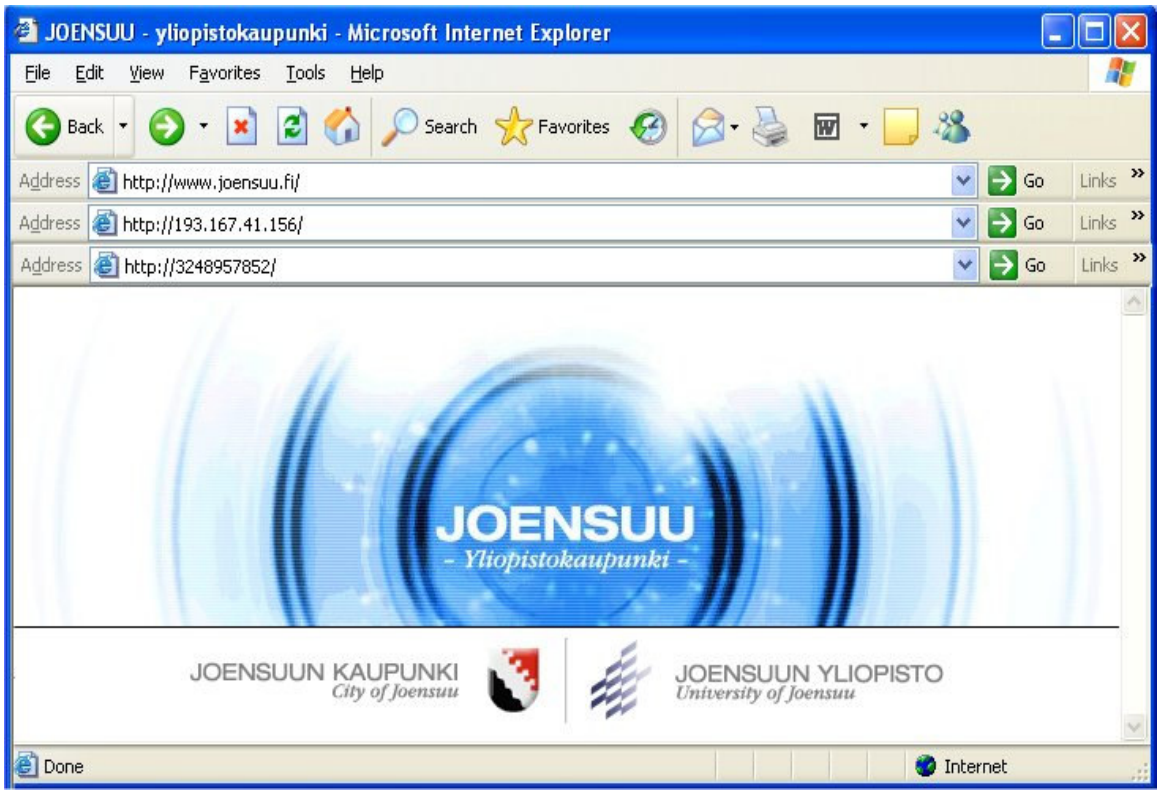
Hoglundin ja McGrawn (2004) mukaan skriptejä voidaan käyttää esimerkiksi ristiinskriptauksessa (cross-site scripting, XSS), jossa tarkoituksena on saada käyttäjän ohjelma suorittamaan muualta tulevaa koodia. Ristiinskriptausta käytetään hyvin yleisesti haittaohjelmien levitykseen. Esimerkiksi keskustelualueen tapauksessa ristiinskriptaus on mahdollista toteuttaa lähettämällä hyökkäyksen kohteelle sähköpostia, joka sisältää vihamielisen, mutta ulkoisesti mahdollisimman luotettavan näköisen linkin haavoittuvuudesta kärsivän keskustelualueen palvelimelle. Hyökkäyksen kohteen käyttäessä saamaansa linkkiä, linkin sisältämä vihamielinen viesti lähetetään palvelimelle. Viestin lähetyksen jälkeen keskustelupalvelin yleensä palauttaa lähetetyn viestin takaisin lähettäjälle, mikä ristiinskriptauksen tapauksessa saa aikaan viestin sisältämän vihamielisen skriptin suorittamisen viestin lähettäjän koneella. Howardin ja LeBlancin (2004) mukaan ongelmaa ei täysin poista pelkästään näennäisesti turvallisten HTML-rakenteiden salliminen, koska hyökkääjä voi upottaa vihamielisen skriptin jonkin palvelimen hyväksymän tunnisteen sisään kuvan 5 esimerkin mukaisesti.

2.3 Vaihtoehtoiset IP-osoitteet

Vaihtoehtoisten koodaustapojen käyttö IP-osoitteiden tapauksessa voi saada aikaan suodatus- ja myös muita tietoturvaongelmia. Tyypilliset vaihtoehtoista koodaustapaa hyödyntävät

hyökkäykset on mahdollista ohjata myös IP-numeroita vastaan. IP-osoite voidaan esittää käyttämällä IP-numeron perusteella muodostettua numerosarjaa (Howard & LeBlanc, 2003). Esimerkiksi jos IP-numero on aaa.bbb.ccc.ddd, niin numerosarja muodostetaan seuraavasti:

```
aaa * 2^24 + bbb * 2^16 + ccc * 2^8 + ddd
```



Kuva 6: Esimerkki vaihtoehtoisista verkkosivun osoitteiden esitystavoista.

Kuvassa 6 on esitetty yhden verkkosivun vaihtoehtoisia esitystapoja. IP-osoitteiden tapauksessa on myös mahdollista käyttää yksittäisten merkkien vaihtoehtoisia koodaustapoja, joiden avulla pyritään välttämään suodatusta. Esimerkiksi osoite

```
http://cs.joensuu.fi/changes.htm
```

voidaan esittää myös seuraavanlaisilla tavoilla:

```
http://cs.joensuu.fi/changes%2Ehtm
http://193.167.42.127/%63%68%61%6E%67%65%73%2E%68%74%6D
http://193.167.42.127:80/changes.htm
```

Esimerkin ensimmäisessä vaihtoehdossa käytetty %2E on heksaesityksen avulla luotu vaihtoehtoinen esitystapa piste-esitykselle. Samoin toisessa vaihtoehdossa tiedostonimi changes.htm on esitetty kokonaan heksaesityksen avulla. Viimeisessä vaihtoehdossa IP-numeron jälkeen lisätty :80 tarkoittaa porttia, johon yhteys otetaan ja portti 80 on TCP-portti, joka on varattu HTTP-protokollalle.

Howardin ja LeBlancin (2004) mukaan erimuotoiset, mutta samaa asiaa tarkoittavat esitykset tulisi muuttaa yhdeksi standardiesitykseksi. Tätä toimenpidettä kutsutaan kanonisoinniksi, jonka tarkoituksena on vähentää samaa tarkoittavien, mutta erilaisten esitysten käyttämisestä johtuvia tietoturvaongelmia.

2.4 Muuttujien ja tunnisteiden ylivuoto

Beattien & al. (2000) mukaan ylivuodon tapauksessa hyökkääjän tarkoituksena on saada haavoittuvan ohjelmiston hallinta itselleen vuodattamalla muistipinoa yli niin, että ohjelmisto saadaan suorittamaan hyökkääjän haluamat toimenpiteet. Ylivuodosta johtuva ohjelmiston virheellinen toiminta voi mahdollistaa esimerkiksi järjestelmätason komentojen suorittamisen ylivuodosta kärsineen ohjelmiston oikeuksilla. McGrawn ja Viegan (2002) mukaan muuttujien ja tunnisteiden ylivuodon tapauksessa hyökkääjän tarkoituksena on syöttää ohjelmalle liian pitkiä merkkijonoja, jotka aiheuttavat haavoittuvassa ohjelmassa ylivuodon.

Tarkastellaan yksinkertaista C-kielellä muodostettua esimerkkiä merkkijonon tapauksessa tapahtuvasta ylivuodosta kuvan 7 mukaisesti. Esimerkissä oleva pääohjelma kutsuu tervehdysaliohjelmaa. Aliohjelma pyytää käyttäjää syöttämään etu- ja sukunimen, jonka jälkeen se tulostaa näytölle tekstin, joka koostuu käyttäjän syöttämästä etu- ja sukunimestä sekä aliohjelmalle parametreina syötetyistä merkkijonoista. Lisäksi pääohjelma tulostaa käyttäjälle tiedon onnistuneesta paluusta aliohjelmasta takaisin pääohjelmaan.

```
#include <stdio.h>

int tervehdys(char *terve, char *viesti)
)
{
        char enimi[8];
        char snimi[8];

        printf("Etunimesi? ");
        gets(enimi);
        printf("Sukunimesi? ");
        gets(snimi);
        printf("%s %s %s, %s\n", terve, enimi, snimi, viesti);
        return 0;
}

int main()
{
        tervehdys("Terve", "varo puskurin ylivuotoa!");
        printf("Takaisin pääohjelmassa\n");
        return 0;
}
```

Kuva 7: Ylivuotohaavoittuvuudesta kärsivä C-kielinen ohjelma.

Kuvassa 7 enimi- ja snimi-muuttujien kooksi on määritelty kahdeksan tavua, mutta käyttäjän antaman syöteen pituutta ei ole tarkastettu. Sopivien syötteiden avulla voidaan esimerkkiohjelman pinoa vuodattaa yli. Pinon ylivuoto tapahtuu esimerkiksi syöttämällä etunimeksi `Maija` ja sukunimeksi `Mainio, Matti`, jolloin ohjelma toimii seuraavanlaisesti:

```
Etunimesi? Maija
Sukunimesi? Mainio, Matti
Terve Matti Mainio, Matti, varo puskurin ylivuotoa!
Takaisin pääohjelmassa
```

Taulukko 2: Esimerkkiohjelman muistipino.

Muistiosoite	Koko	Sisältö
ffd4 (65492)	2 tavua	osoitin *viesti*-muuttujaan
ffd2 (65490)	2 tavua	osoitin *terve*-muuttujaan
ffd0 (65488)	2 tavua	paluuosoite
ffce (65486)	2 tavua	ympäristö/kehysosoitin (frame pointer)
ffc6 (65478)	8 tavua	*enimi*-muuttuja
ffbe (65470)	8 tavua	*snimi*-muuttuja

Taulukko 2 esittää ohjelman muistipinon rakenteen. Ympäristöosoitin osoittaa aliohjelmien omien paikallisten muuttujien alkamisosoiteeseen, eli tässä tapauksessa ympäristöosoitin osoittaa muistiosoitteeseen ffbe. Kuten taulukosta 2 voidaan havaita, ohjelmalle syötetty liian pitkä snimi-muuttujan arvo ylikirjoittaa enimi-muuttujan arvon, jolloin snimi-muuttujan loppuosa tulee enimi-muuttujan arvoksi. Kuva 8 esittää ylivuodon tapahtumisen.

	ffbe							ffc5	ffc6							ffcd
enimi									M	a	i	j	a			
snimi	M	a	i	n	i	o	,		M	a	t	t	i			

Kuva 8: Muuttujien varaama tila.

Tarpeeksi pitkällä enimi- tai snimi-muuttujan arvolla ohjelmassa tapahtuu segmenttivirhe, joka johtuu liian pitkän muuttujan arvon ylikirjoittaessa pinossa olevan paluuosoitteen päälle. Tässä tilanteessa aliohjelma kirjoittaa tulosteen normaalisti, mutta kaatuu ennen ”Takaisin pääohjelmassa”–rivin tulostusta. Ohjelma toimii siis normaalisti siihen asti, kunnes pyritään palaamaan takaisin pääohjelmaan tervehdys-aliohjelmasta. Jos hyökkääjä pystyy muuttamaan paluuosoitteen haluamakseen, niin on mahdollista pakottaa kohdeohjelma suorittamaan hyökkääjän haluamaa ohjelmakoodia.

3. Turvallisuuden suunnittelu uhkamallinnuksen avulla

Tietoturvauhkamallinnus (security threat modeling) on tietoturvapohjainen riskianalyysi, joka auttaa määrittelemään järjestelmää vakavimmin uhkaavat tietoturvariskit ja sen kuinka hyökkäykset voidaan toteuttaa. Ryhmä, joka tekee uhkamallin sovelluksesta, oppii ymmärtämään sen, mistä järjestelmä on kaikkein haavoittuvaisin ja osaa valita sopivan tekniikan uhkan lieventämiseksi (Howard ja LeBlanc, 2003).

Howardin ja LeBlancin (2003) tutkimuksen perusteella suuri osa ohjelmistoista on suunniteltu ilman selkeää menetelmää ja turvallisuus on usein tällaisen kaaoksen uhri. Yksi tapa tuoda suunnitelmallisuutta suunnitteluvaiheeseen on luoda uhkamalli. Uhkamalli auttaa ymmärtämään mistä kohtaa sovellus on haavoittuvaisin ja tämän ansiosta on helpompi valita oikeat työkalut turvallisempien järjestelmien toteuttamiseksi.

Uhkien analysointi voi olla iso työ, mutta on oleellisen tärkeää käyttää siihen aikaa. On halvempaa etsiä turvallisuuden sunnitteluvirhe suunitteluvaiheessa ja ratkaista se ennen kuin ohjelmointi alkaa. Uhkamallin luontiprosessi etenee uhkamallinustiimin kokoamisen jälkeen seuraavasti (Howard ja LeBlanc, 2003):

1. Ositetaan sovellus.
2. Määritellään järjestelmää koskevat uhat.
3. Asetetaan uhat järjestykseen riskin mukaan.
4. Valitaan tapa vastata kuhunkin uhkaan.
5. Selvitetään eri tekniikat, joilla uhkaa voidaan lieventää.
6. Valitaan sopivin tapa löydetyistä tekniikoista.

Yleensä osa tämän prosessin vaiheista joudutaan toteuttamaan useamman kerran, koska on vaikeaa löytää kaikki sovellusta koskevat uhat kerralla. Myös muutoksia tapahtuu kaiken aikaa, löytyy uusia uhkia, uusia asioita opitaan ja liiketoiminta- ja tekninen ympäristö muuttuvat. Kaikilla näillä on vaikutuksensa järjestelmän uhkiin.

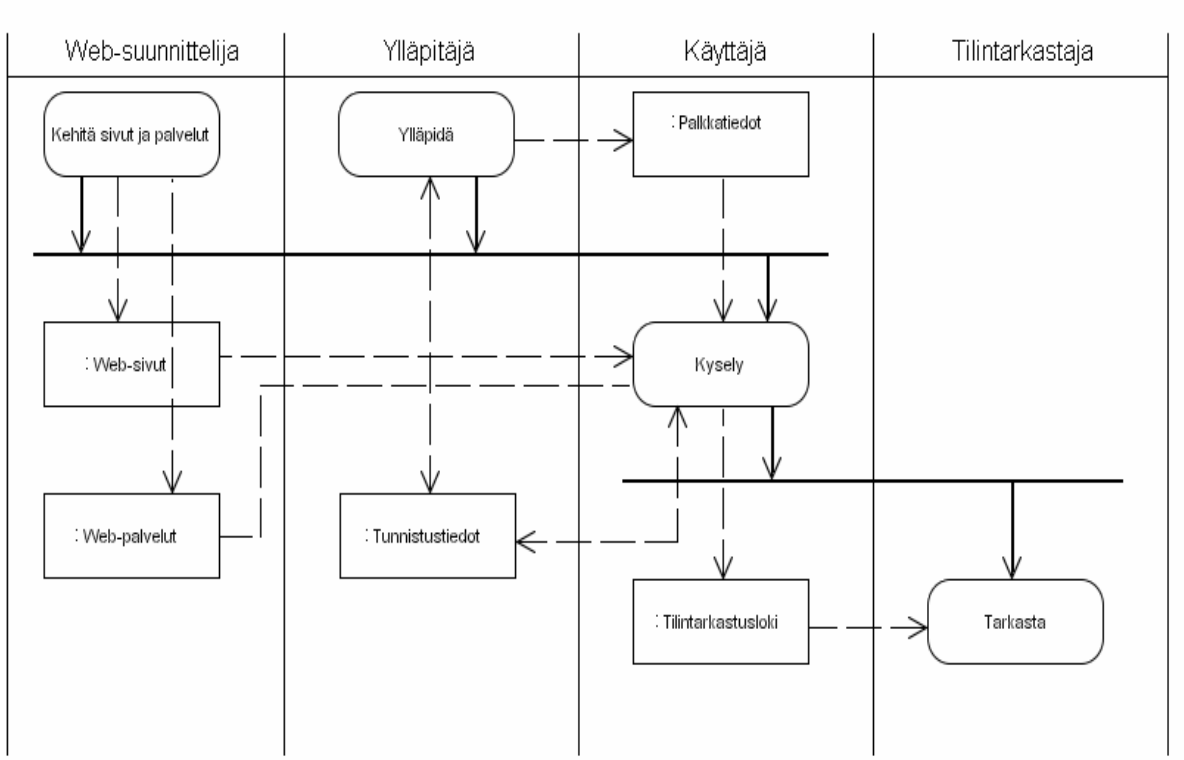
3.1 Ohjelmiston jakaminen osiin

Howard ja LeBlanc (2003) määrittelevät ohjelmiston osiin jakamisen tiedonkeruuvaiheeksi, jonka tavoitteena on ymmärtää ohjelmiston rakenne, järjestelmän vaatimukset ja ehdotettu arkkitehtuuri korkealla tasolla. McGrawn ja Viegan (2002) mukaan tiedonkeruuvaiheessa on tärkeää olla hyvin dokumentoidut tietoturvavaatimukset, jotka kuvaavat järjestelmän tietoturvamenettelytavat. Ilman kuvaa turvallisuusmenettelytavoista työ vaikeutuu, koska suunnittelija joutuu tekemään oletuksia turvallisuusvaatimuksista.

Ensimmäinen vaihe ohjelmiston osittamisessa on määritellä analysoitavan järjestelmän rajat tai soveltamisalue sekä ymmärtää ohjelmiston luotettujen ja epäluotettavien, kuten esimerkiksi kolmannen osapuolen toimittamien komponenttien väliset rajat. Jos ohjelmistolle ei määritellä soveltamisaluetta, niin sen seurauksena kulutetaan huomattavasti aikaa uhkiin, jotka ovat soveltamisalueen ulkopuolella ja ohjelmiston kontrollin ulottumattomissa (Howard ja LeBlanc, 2003).

Ohjelmiston osiin jakamisen apuna voidaan käyttää esimerkiksi rakenteisen analyysin *tietovuokaavioita* (data flow diagram), mutta myös UML-mallinnuskielellä (Unified Modeling Language) tehty kuvaus ajaa saman asian (Howard ja LeBlanc, 2003). Nykyisin olioparadigmaa sovelletaan yleisesti eikä käsittääkseni olioperustaisen järjestelmän mallintaminen rakenteisen analyysin menetelmin ole oikea tapa. Tästä syystä pyrin muuttamaan Howardin ja LeBlancin (2003) käyttämän notaation UML-notaation mukaiseksi. UML-kuvauksissa käytän UML:n versiota 1.5 (Object Management Group, 2005).

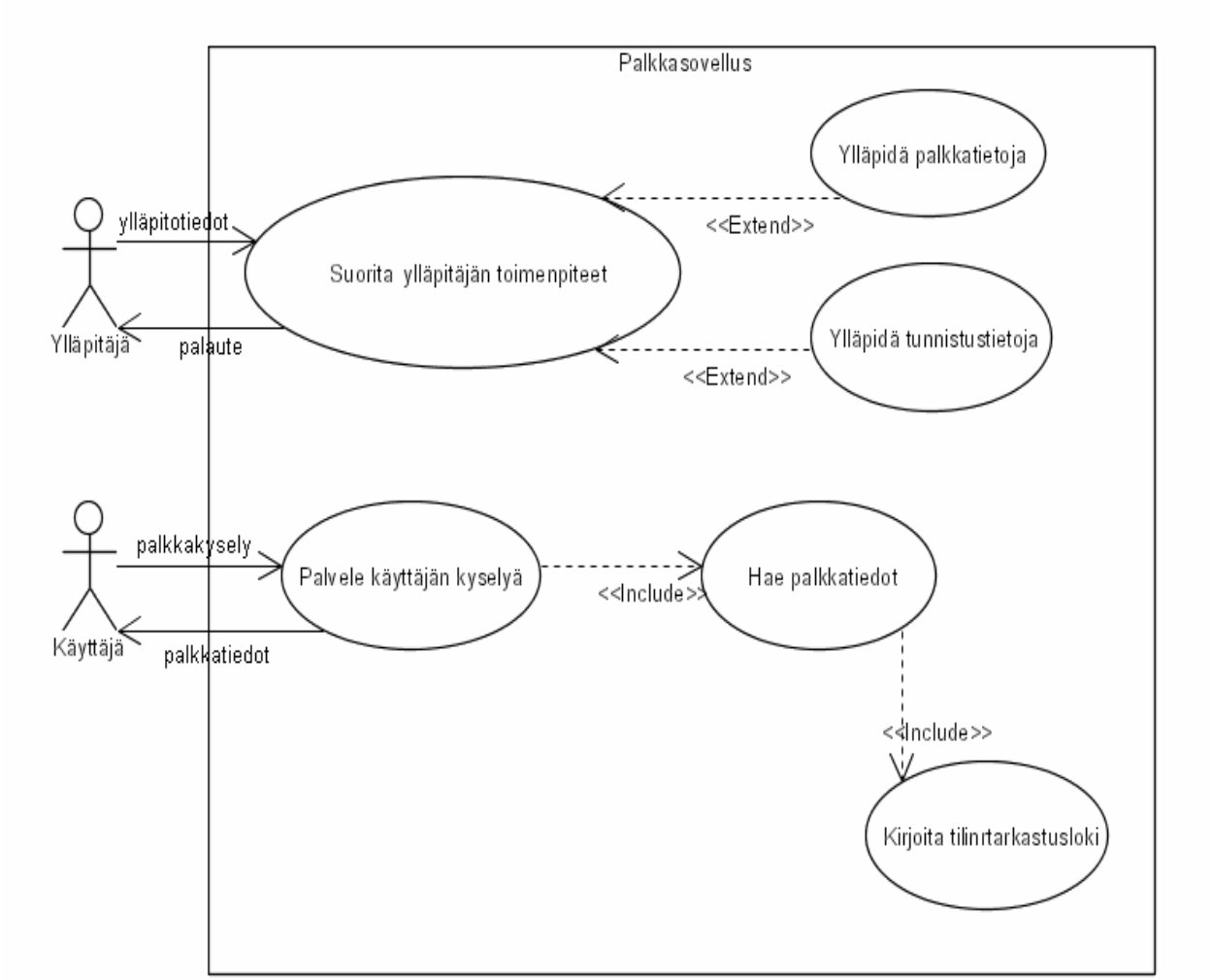
Pääperiaate UML-kuvauksen avulla ohjelmiston osiin jakamisessa on jakaa tarkasteltava järjestelmä alijärjestelmiin ja tarkentaa järjestelmää eri kaaviotekniikoilla. Tarkastellaan esimerkkinä yksinkertaista web-perustaista palkkasovellusta, jonka avulla työntekijät voivat tutkia palkkatietojaan. Kuva 9 esittää palkkasovelluksesta luotua kohdealueen liiketoiminnan toimintokaaviota.



Kuva 9: Kohdealueen liiketoiminnan kuvaus toimintokaavion avulla.

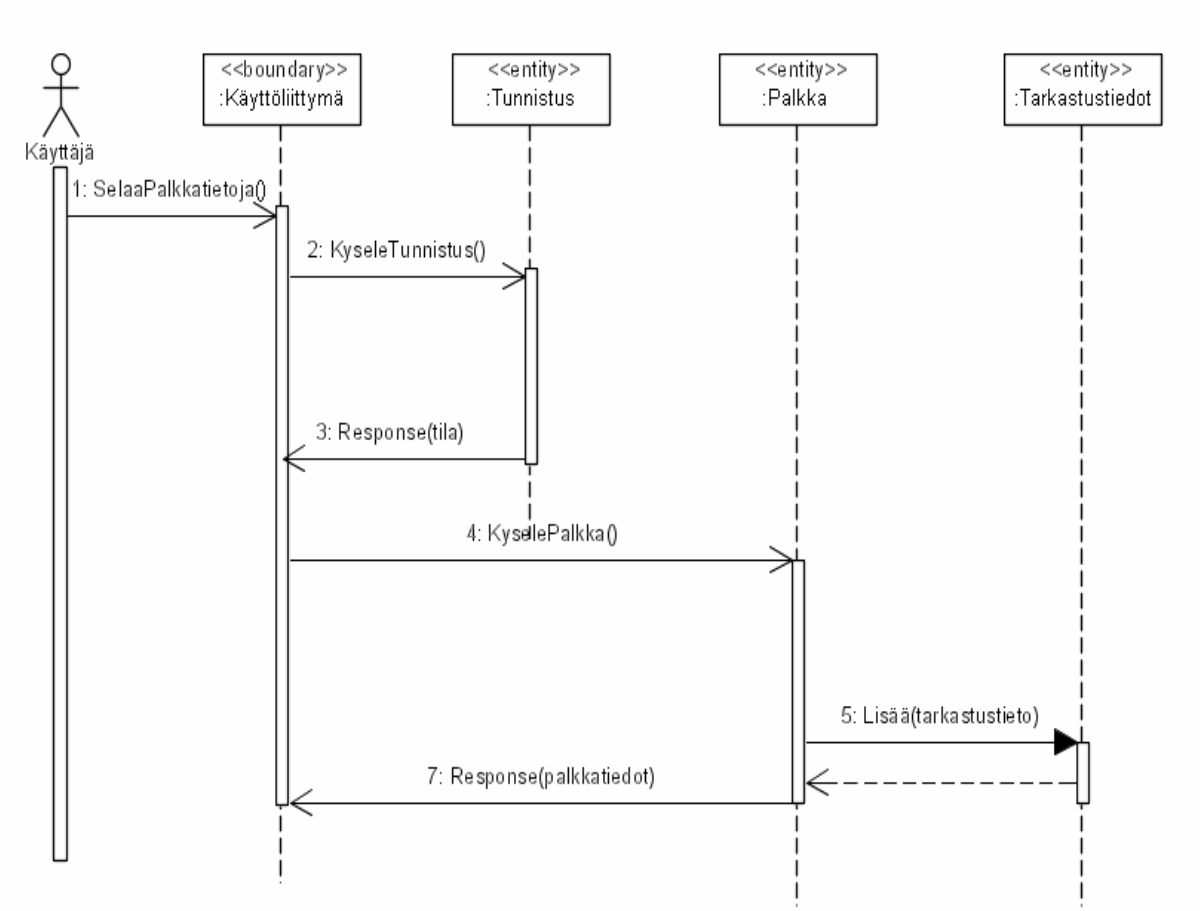
Kohdealueen liiketoiminnan kuvaus sisältää kohdealueen toimijat, jotka palkkasovelluksen tapauksessa ovat web-suunnittelija, ylläpitäjä, käyttäjä ja tilintarkastaja. Web-suunnittelijan tehtävä on kehittää palkkasovelluksen käyttämät web-sivut ja –palvelut. Ylläpitäjä ylläpitää tunnistus- ja palkkatietoja. Web-sivuja ja –palveluita sekä tunnistus- ja palkkatietoja tarvitaan käyttäjän suorittamassa palkkatietojen kyselyssä. Käyttäjä suorittaa kyselyitä, joiden seurauksena luodut tilintarkastustiedot päätyvät tilitarkastajalle tarkastettaviksi. Toimintokaavioon on myös merkitty toimintovuon yhdistyminen ja haarautuminen, sekä oliovuot.

Toimintokaaviosta voidaan luontevasti siiryä tarkastelemaan käyttötapauskaaviota käyttäjän ja ylläpitäjän osalta. Käyttötapauskaavio esittää järjestelmän toiminallisuutta edustavien käyttötapauksien ja järjestelmää käyttävien toimijoiden väliset suhteet (Object Management Group, 2005). Esittämäni käyttötapauskaavio ei sisällä web-suunnittelijaa eikä tilintarkastajaa, koska näiden toimijoiden suorittamat tehtävät eivät ole palkkasovelluksen sisäisiä, vaan ne tapahtuvat sovelluksen ulkopuolella.



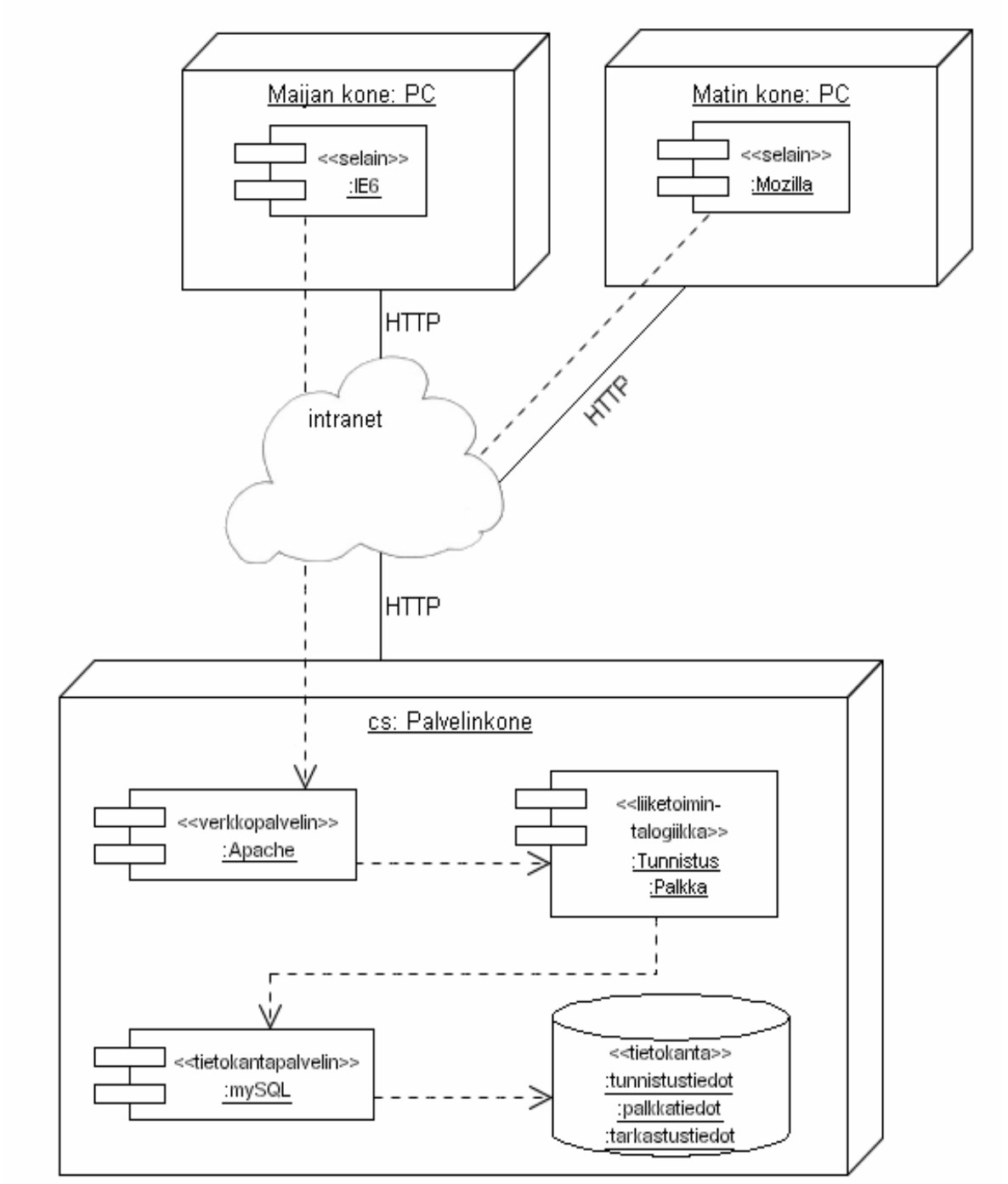
Kuva 10: Palkkasovelluksen käyttötapauskaavio.

Tarkentamalla kuvassa 10 esitettyä käyttötapauskaaviota molempien toimijoiden osalta, voidaan luoda sekvenssikaavioita, jotka kuvaavat järjestelmässä tapahtuvia toimenpiteitä (Object Management Group, 2005). Näiden toimenpidesekvenssien avulla voidaan määritellä järjestelmää koskevat uhat. Kuvassa 11 esitetty sekvenssikaavio kuvaa järjestelmässä tapahtuvia toimenpiteitä käyttäjän kysellessä omia palkkatietojaan.



Kuva 11: Palkkasovelluksen sekvenssikaavio palkkatietojen selaamiseksi.

Ohjelmiston osiin jakamisen seurauksena on mahdollista hahmotella järjestelmän yleisarkkitehtuuria. Kuvassa 12 on sijoittelukaavion avulla esitetty tietoverkon huomioiva palkkasovelluksen yleisarkkitehtuuri käyttäjän suorittamien toimintojen osalta. Sijoittelukaavio kuvaa järjestelmään kuuluvat ohjelmistokomponentit, prosessit ja objektit (Object Management Group, 2005). Abstraktimpi tapa olisi kuvata arkkitehtuuria pakettien, luokkien ja rajapintojen avulla.



Kuva 12: Palkkasovelluksen sijoittelukaavio käyttäjän osalta.

Seuraava vaihe on määritellä yleisarkkitehtuurin perusteella sovelluksen ydinprosessit, tallennettavat ja tilapäiset tiedot ja tiedonsiirtoprotokollat. Kuvien 11 ja 12 esittämästä järjestelmästä on mahdollista tunnistaa taulukon 3 mukaiset komponentit ja protokollat.

Taulukko 3: Web-perustaisen sovelluksen käyttäjän toimintoihin liittyvät komponentit ja protokollat.

Komponentti tai protokolla	**Kommentti**
Käyttäjä	Käyttäjät ovat tämän ratkaisun pääasiakkaita. Järjestelmän avulla he voivat tarkastella palkkatietojaan.
Käyttöliittymä	Käyttöliittymä on HTML-perustainen ja se on käyttäjien ensisijainen tapa päästä järjestelmään.
Verkkopalvelin	Verkkopalvelin muodostaa HTML-muotoisen tiedon, jonka käyttäjän selain esittää.
Verkkosivut	Liiketoimintalogiikkaa hyödyntävät verkkosivut esittävät käyttäjän palkkatiedot.
Liiketoiminta-logiikka	Liiketoimintalogiikka-komponentti ottaa vastaan käyttäjien kyselyt ja määritelee käyttäjälle näytettävät asiaankuuluvat tiedot.
Tietokantapalvelin	Tietokantapalvelin käsittelee palkkatietoja ja luo tarkastustiedot
Tunnistustiedot	Tunnistustietoja käytetään määrittelemään ovatko käyttäjät niitä, joita he väittävät olevansa.
Palkkatiedot	Palkkatiedot sisältävät työntekijän palkkatietojen historian.
Tarkastustiedot	Tietokantapalvelin muodostaa tarkastustiedon, kun joku muodostaa, lukee, päivittää ja poistaa palkkatietoja.
HTTP	HTTP on protokolla, jota HTML-perustainen käyttöliittymä ja verkkopalvelin käyttävät tiedonsiirtoon.

Howardin ja LeBlanc (2003) mukaan ohjelmiston osiin jakamisessa tulisi päästä tilanteeseen, jossa ymmärretään ohjelmiston rakenne. Osiin jakamisessa tarvitsee mennä kuitenkin vain niin syvälle, että on mahdollista ymmärtää ohjelmiston vaarana olevat uhat.

3.2 Järjestelmän uhkien määrittely

Uhkamallinnuksen seuraava vaihe on käyttää taulukossa 3 kuvattuja komponentteja uhkien kohteina. Käsiteltäessä uhkia on käytännöllistä tarkastella jokaista sovelluksen komponenttia ja kysyä seuraavan kaltaisia kysymyksiä:

- Voiko valtuuttamaton käyttäjä nähdä luottamuksellisia verkon tietoja?
- Voiko ei-luotettu käyttäjä muokata tietokannan tietoja?
- Voiko joku estää validia käyttäjää käyttämästä sovellusta?
- Voiko jotain sovelluksen ominaisuutta käyttää oikeuksien korottamiseen?

3.2.1 STRIDE-mallin käyttö uhkien ryhmittelyyn

Järjestelmänkomponentteihin kohdistuvien kysymysten esittämisen apuna tulisi käyttää uhkien ryhmittelyä, esimerkiksi Howardin ja LeBlancin (2003) esittelemää STRIDE-uhkamallia. STRIDE on lyhenne, joka muodostuu seuraavista kuudesta uhkaryhmästä:

S *Identiteetin väärentäminen* (Spoofing identity). Väärennetyllä identiteetillä petkuttaminen tapahtuu silloin, kun luvatta selvitetään ja sen jälkeen käytetään toisen käyttäjän tunnistustietoja, kuten käyttäjätunnusta ja salasanaa.

T *Tietojen sekoittaminen* (Tampering with data), joka tarkoittaa tietojen vihamielistä muuttamista. Esimerkkinä tästä voi mainita luvattomasti tehdyt muutokset pysyvästi tallennettuihin tietoihin, kuten tietokannan tietoihin, ja tietojen muuttaminen, kun ne siirtyvät kahden koneen välillä Internetin tapaisessa avoimessa verkossa.

R *Torjuminen* (Repudiation) liittyy käyttäjiin, jotka kieltävät suorittaneensa toiminnon. Esimerkiksi käyttäjä suorittaa laittoman toiminnon järjestelmässä, joka ei kykene jäljittämään kiellettyjä toimintoja. Torjunnan estäminen on järjestelmän kyky vastata torjuntauhkaan.

I *Tietojen paljastaminen* (Information disclosure) sisältää tietojen esittämisen henkilöille, joilla ei pitäisi olla siihen oikeutta. Esimerkiksi käyttäjän mahdollisuus lukea tiedos-

toa, johon hänelle ei ole myönnetty oikeutta, ja tunkeilijan mahdollisuus lukea tietoja kahden koneen välisen siirron aikana.

D *Palvelun esto* (Denial of service, DoS) estää palvelun käyttämisen siihen oikeutetuilta käyttäjiltä esimerkiksi tekemällä web-palvelusta tilapäisesti käyttökelvottoman. On tärkeää suojautua määrätyn tyyppisiä DoS-uhkia vastaan, että voidaan parantaa järjestelmän käytettävyyttä ja luotettavuutta.

E *Oikeuden nostaminen* (Elevation of privilege). Tämän tyyppisessä uhassa käyttäjä, jolle ei ole virallisesti myönnetty oikeuksia, hankkii niitä ja kykenee sen jälkeen häiritsemään tai tuhoamaan koko järjestelmän. Oikeuden nostamisen uhka sisältyy tilanteisiin, joissa hyökkääjä on murtautunut kaikkien turvajärjestelmien läpi ja tullut osaksi luotettua järjestelmää.

Joillakin uhkatyypeistä voi olla keskinäinen suhde. Ei ole ollenkaan mahdotonta, että tietojen paljastamisesta johtuva uhka voi johtaa identiteetin väärentämiseen, esimerkiksi jos käyttäjien käyttäjätietoja ei ole suojattu. Oikeuden nostaminen on pahin mahdollinen uhka, koska pystyttäessä hankkimaan pääkäyttäjän oikeudet järjestelmään kaikki STRIDE-mallin mukaiset uhat ovat todellisia.

Seuraava tehtävä on tunnistaa järjestelmää koskevat uhat. Yksinkertainen tapa soveltaa STRIDE-mallia sovellukseen on tutkia, miten kukin mallin uhkista vaikuttaa kuhunkin sovelluksen komponenttiin ja sen suhteisiin muihin ratkaisun komponentteihin. Taulukossa 4 esitetään, kuinka tietynlaiset STRIDE-mallin mukaiset uhkatyypit vaikuttavat eri kohteisiin.

Taulukko 4: Uhkatyypin vaikutus eri kohteisiin (Howard ja LeBlanc, 2003).

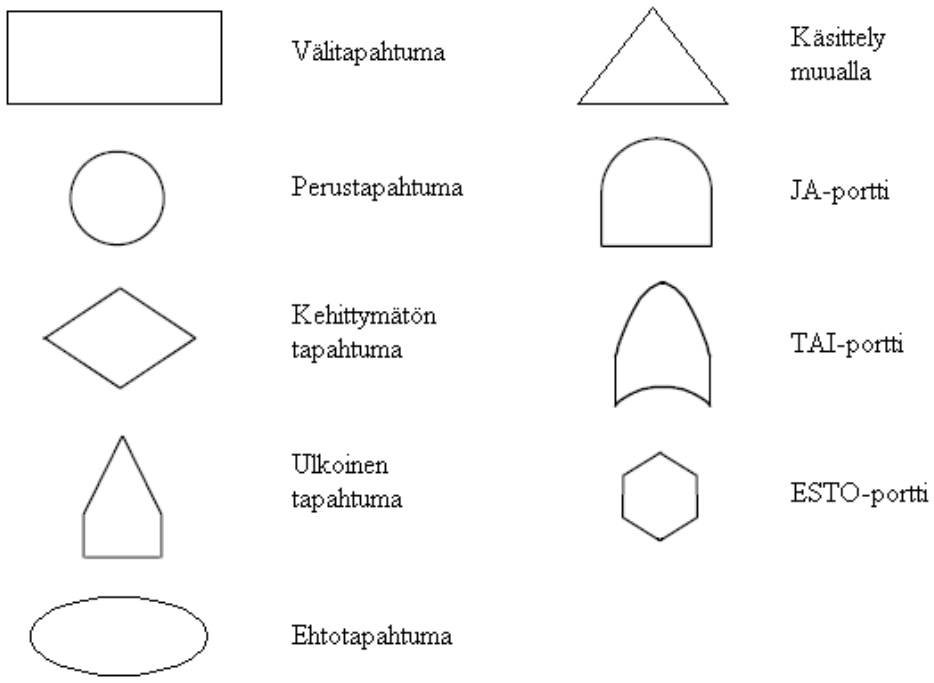
Uhan tyyppi	Vaikuttaa prosesseihin	Vaikuttaa tietovarastoihin	Vaikuttaa toimijoihin	Vaikuttaa tietovuohon
S	K		K	
T	K	K		K
R		K	K	K
I	K	K		K
D	K	K		K
E	K			

Uhkia tunnistettaessa sovelluksen komponentit ja prosessit tulisi McGrawn ja Viegan (2002) mukaan tutkia prioriteettijärjestyksessä ja Howardin ja LeBlancin (2003) mukaan tulisi tutkia koskeeko jokin mallin S-, T-, R-, I-, D- tai E–osa tutkittavana olevaa komponenttia tai prosessia. On mahdollista, että useampia osia koskee useita uhkia ja on tärkeää tunnistaa ne kaikki (Howard ja LeBlanc, 2004). Tässä vaiheessa tulee tarkastella myös ulkopuolisilta toimittajilta tulleita järjestelmän osia, sillä ne muodostavat myös todellisen uhan (McGraw ja Viega, 2002). Howardin ja LeBlancin (2003) mukaan järjestelmän uhkien tunnistamisessa voidaan käyttää apuna myös hyökkäyspuita ja soveltaa STRIDE-mallia hyökkäyspuihin.

3.2.2 Hyökkäyspuun käyttö järjestelmän uhkien määrittelyssä

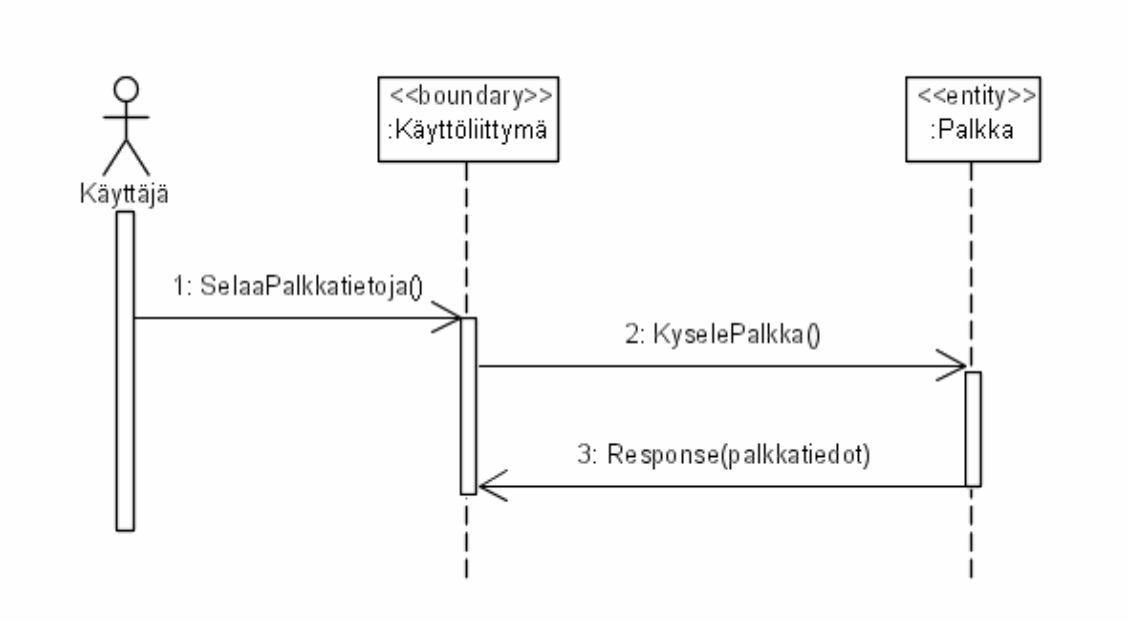
Järjestelmällinen tapa järjestelmän uhkien tunnistamiseksi on rakentaa *hyökkäyspuu* (attack tree). Hyökkäyspuun avulla voidaan testata onko sovellus haavoittuvainen esimerkiksi hyökkäysmallin esittämälle hyökkäystekniikalle. Ajatuksena on luoda kaavio, joka kuvaa hyökkääjien päätöksentekoprosessia. Puun juuri kuvaa hyökkääjien mahdollista päämäärää ja lehdet kuvaavat tapoja halutun päämäärän saavuttamiseksi. Mitä alemmaksi puussa edetään, sitä tarkemmaksi hyökkäys etenee (McGraw ja Viega, 2002). Tällaista kuvaustapaa kutsutaan jäsentäväksi eli deduktiiviseksi menetelmäksi.

Hyökkäyspuun tarkoituksena on siis esittää ehtojen joukko, joiden tulee olla tosia, että hyökkäys voidaan toteuttaa kohdesovelluksessa. Kuvassa 13 on esitetty U.S. Nuclear Regulatory Commissionin (1981) NUREG-0492-käsikirjaan perustuvia yleisimpiä hyökkäyspuun mallintamiseen soveltuvia symboleita.



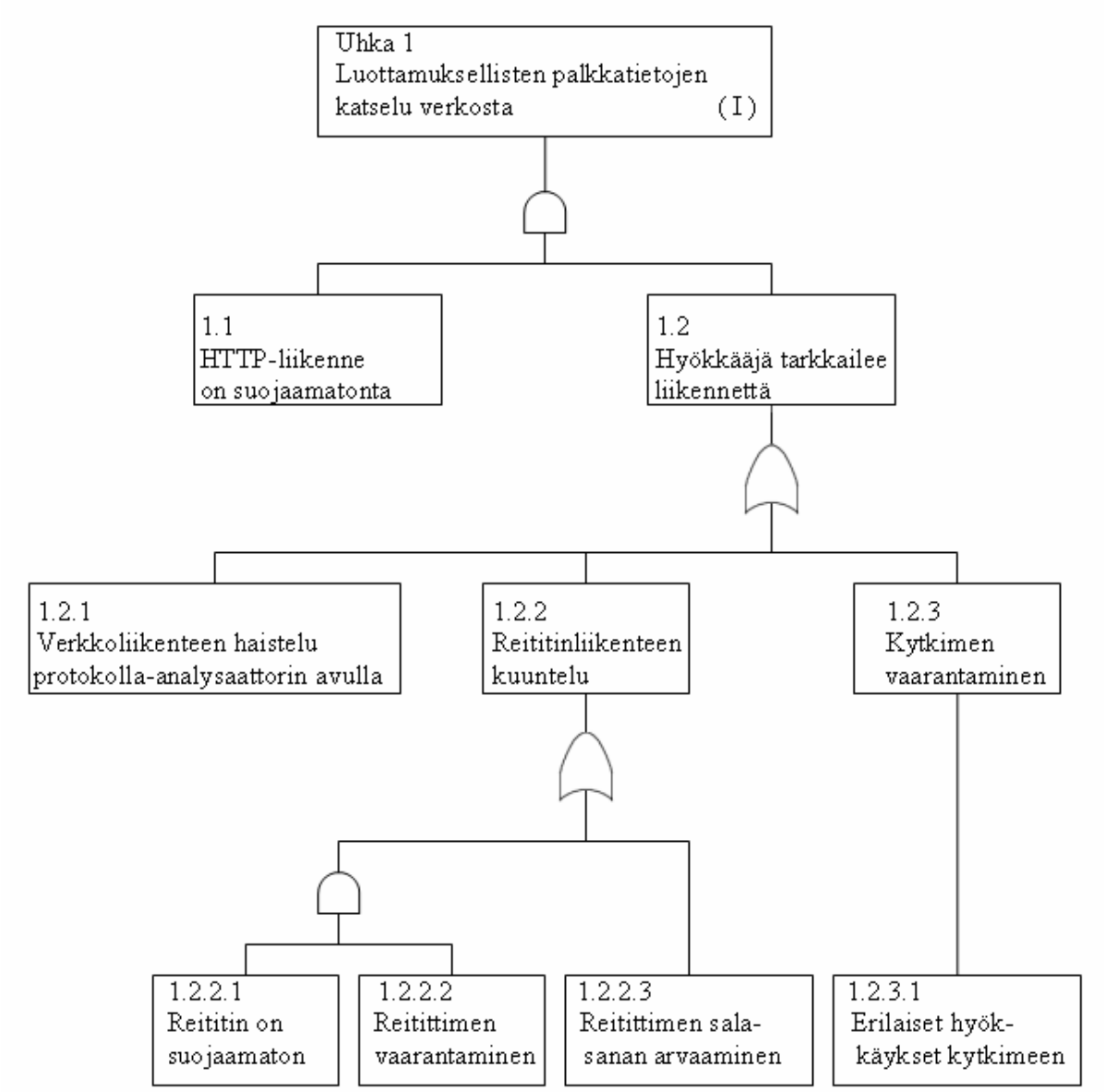
Kuva 13: Hyökkäyspuissa käytettyjä symboleja (U.S. Nuclear Regulatory Commission, 1981).

Tarkastellaan yksinkertaista hyökkäyspuuta, joka on muodostettu palkkasovelluksesta. Kuva 14 esittää kuvassa 11 esitetyn sekvenssikaavion osaa, jossa kuvataan palkkatietojen lähettämistä verkkopalvelimelta käyttäjän koneelle.



Kuva 14: Sekvenssikaavio verkkopalvelimen ja käyttäjän välillä tapahtuvista toimenpiteistä.

Tutkitaan miten hyökkääjä voi tarkastella käyttäjän palkkatietoja, kun palkkatiedot siirtyvät palvelimelta käyttäjän koneelle. Uhan muodostumista tutkitaan muodostamalla kyseisestä tapahtumasta hyökkäyspuu kuvan 15 mukaisesti.



Kuva 15: Hyökkäyspuu (Howard ja LeBlanc, 2003).

Kuvan 15 hyökkäyspuun ylin laatikko on lopullinen uhka eli juuri ja laatikot sen alla ovat askeleita, joiden avulla uhka on mahdollista toteuttaa. Kyseisen uhan tapauksessa STRIDE-mallin mukaisesti on kyse I-tyyppisestä uhasta eli tietojen paljastamisesta. Ennen kuin uhkaa on mahdollista hyödyntää, tulee HTTP-liikenteen olla suojaamatonta (1.1) ja hyökkääjän tu-

lee pystyä tarkkailemaan verkossa tapahtuvaa liikennettä (1.2). Jotta hyökkääjän olisi mahdollista tarkkailla verkkoliikennettä, hänen täytyy ”haistella” verkkoa (1.2.1) tai kuunnella reitittimen (1.2.2) tai kytkimen (1.2.3) läpi kulkevia tietoja. Koska kyseisessä tapauksessa verkkoliikenne on suojaamatonta, niin hyökkääjä voi katsella tietoja tapauksissa (1.2.1) ja (1.2.2). Voidakseen kuunnella reititinliikennettä täytyy reitittimen olla suojaamaton (1.2.2.1) ja sen toiminta tulee pystyä vaarantamaan (1.2.2.2), tai hyökkääjän tulee pystyä arvaamaan tai muulla tavoin hankkimaan reitittimen ylläpitäjän salasana (1.2.2.3). Hyökkäyspuu voidaan esittää suppeammin tekstimuodossa (Howard ja LeBlanc, 2003) kuvan 16 mukaisesti.

```
1.0 Luottamuksellisten palkkatietojen katselu verkosta
  1.1 HTTP-liikenne on suojaamatonta (JA)
    1.2 Hyökkääjä tarkkailee verkkoliikennettä
      1.2.1 Verkkoliikenteen haistelu protokolla-
            analysaattorin avulla
      1.2.2 Reititinliikenteen kuuntelu
          1.2.2.1 Reititin on suojaamaton (JA)
            1.2.2.2 Reititin on vaarannettavissa
          1.2.2.3 Reitittimen salasanan arvaaminen
      1.2.3 Kytkimen vaarantaminen
          1.2.3.1 Erilaiset kytkimeen kohdistuvat
                  hyökkäykset
```

Kuva 16: Hyökkäyspuun tekstiesitys.

Pienten muutosten avulla hyökkäyspuista on mahdollista tehdä luettavampia (Howard ja LeBlanc, 2003). Luettavuutta voidaan parantaa käyttämällä pisteviivoja osoittamaan epätodennäköisiä hyökkäyksiä ja yhtenäisiä viivoja osoittamaan todennäköisempiä hyökkäyksiä. Hyökkäyspuun jokaiseen solmuun on myös mahdollista lisätä arvo, joka kuvaa havaittua riskiä. Riskien arviointia käsitellään tarkemmin alakohdassa 3.3.

3.2.3 Uhkamallinnuksen aikana löydetyistä uhkista kirjattavat asiat

Löydetystä uhasta tulee Howardin ja LeBlancin (2003) mukaan tallentaa muutakin kuin sen nimi ja tyyppi. Jokaisesta uhasta tulisi määritellä ja kirjata taulukossa 5 esitetyt asiat.

Taulukko 5: Uhkamallinnuksen aikana kirjattavat asiat (Howard ja LeBlanc, 2003).

Asia	**Kommentit**
Nimi	Uhan tulee olla selvä nimen perusteella, esimerkiksi ”Hyökkääjä käsittelee käyttäjän ostoskoria.”
Kohde	Mikä sovelluksen osa on hyökkäyksen alla? Esimerkiksi palkkasovelluksen kohteisiin kuuluvat selain, web-palvelin, palkkakomponentti, palkkatietokanta ja niin edelleen
Uhan tyyppi	Uhan tyyppi kirjataan STRIDE-malliin perustuen. Uhka voi kuulua useaan STRIDE-ryhmään
Todennäköisyys	Mikä on todennäköisyys sille, että uhkaan liittyvä hyökkäys tapahtuu? Myös automatisoitavat hyökkäykset tulee käsitellä vastaavasti. Hyökkäyksen mahdollisuus arvioidaan asteikolla 1 (pienin todennäköisyys) – 10 (suurin todennäköisyys)
Kriittisyys	Kriittisyys liittyy hyökkäyksen mahdollisesti aiheuttaman vahingon laajuuteen ja vakavuuteen. Hyökkäyksen kriittisyys arvioidaan asteikolla 1 (pienin mahdollinen vahinko) – 10 (suurin vahingon mahdollisuus).
Hyökkäyspuu	Miten hyökkääjä ilmaisee uhan olemassaolon?
Lievennystekniikat (valinnainen)	Mikä lieventää uhkaa? Tulee myös huomioida, miten hankala uhkaa on lieventää; jotkin uhat ovat helpompia lieventää kuin toiset. Ajatus siitä, miten vaikea uhkaa on lieventää, helpottaa sen priorisoinnissa.
Lieventämisen tila	Mahdolliset tilat ovat: Kyllä, Ei, Hieman ja tarvitaan tutkimuksia
Virheen numero (valinnainen)	Jos käytössä on virheet kirjaavaa tietokanta, niin tietokantaan tulee kirjata virheen numero. Uhkamallinnustietokanta tai –työkalu ei saa korvata virhetietokantaa. Uhasta kerätään sen verran tietoja mitä uhkamallinnusprosessissa tarvitaan ja lisäksi tulee ylläpitää virhetietokantaa.

3.3 Uhkien luokittelu kohoavan riskin mukaan

Uhkapuiden luomisen jälkeen tulisi määritellä kaikkien riskialttiimmat uhat. Tällä tavalla työ voidaan priorisoida (Howard ja LeBlanc, 2003). Helppo tapa kunkin uhan aiheuttaman riskin laskemiseksi on kertoa uhan kriittisyys sen todennäköisyydellä kaavan (1) mukaisesti. Arvo 1 tarkoittaa vähäistä kriittisyyttä tai todennäköisyyttä ja arvo 10 tarkoittaa suurta kriittisyyttä tai todennäköisyyttä.

$$\text{RISKI} = \text{Kriittisyys} * \text{Todennäköisyys} \quad (1)$$

Mitä suurempi tulo saadaan, sitä suuremman kokonaisriskin tutkittavana oleva uhka aiheuttaa järjestelmälle. Toinen tapa uhkien aiheuttamien riskien määrittämiseksi on käyttää Howardin ja LeBlancin (2003) esittelemää *DREAD*-mallia, joka koostuu seuraavista tekijöistä:

D *Vahingon vaikutusmahdollisuus* (Damage potential). Kuinka suuri vahinko voi olla? Uhkaa verrataan mahdollisen todellisen vahingon laajuuteen. Tyypillisesti pahin (arvoltaan 10) on uhka, joka antaa hyökkääjälle mahdollisuuden kiertää kaikki tietoturvarajoitukset ja mahdollistaa kaiken suorittamisen virtuaalisesti. Oikeuden nostamiseen liittyvät uhat ovat yleensä arvoltaan 10.

R *Toistettavuus* (Reproducibility). Kuinka helppoa on saada potentiaalinen uhka toimimaan? Arvo kuvaa sitä, kuinka helppoa uhkaa on hyödyntää. Joitakin ohjelmointivirheitä voidaan hyödyntää aina (arvoltaan 10), mutta toiset, kuten monimutkaiset ajoitukseen perustuvat kilpailutilanteet ovat ennalta-arvaamattomia ja saattavat toimia vain joissakin harvoissa tilanteissa.

E *Hyödynnettävyys* (Exploitability). Kuinka paljon ponnisteluja ja tietämystä tarvitaan hyökkäyksen järjestämiseksi? Esimerkiksi jos aloitteleva ohjelmoija voi suorittaa hyökkäyksen kotitietokoneen avulla, niin arvoksi saadaan 10.

A *Vaikutus käyttäjiin* (Affected users). Hyökkäyksen tapahtuessa, kuinka moneen käyttäjään hyökkäys vaikuttaa? Tämä mittaa karkeasti, kuinka monta prosenttia käyttäjistä joutuu hyökkäyksen kohteeksi: 91-100 % (arvo 10) -> 0-10 % (arvo 1). Täytyy miettiä myös todellista käyttäjämäärää ja suhteuttaa arvio sen mukaiseksi.

D *Löydettävyys* (Discoverability). Tämä on vaikein arvo määrittää. Jos löydettävyyden arvoa ei pystytä määrittelemään, niin arvoksi voidaan laittaa 10 ja luokitella uhat muiden arvojen avulla.

Taulukossa 6 on esitelty karkea määrittely DREAD-mallin arvojen määrittämisestä.

Taulukko 6: DREAD-mallin arvojen määrittely (Small, 2004).

	Vaikutus-mahdollisuus (D)	**Toistettavuus (R)**	**Hyödynnettävyys (E)**	**Vaikutus käyttäjiin (A)**	**Löydettävyys (D)**
Arvo	Millaista tuhoa hyökkäys voi aiheuttaa	Missä laajuudessa hyökkäyksen onnistuminen on determinististä	Kuinka paljon ponnisteluja ja taitoja hyökkäyksen toteuttaminen vaatii	Hyökkäyksestä kärsivien käyttäjien määrä.	Kuinka helposti hyökkäys vodaan havaita
10	Hyökkäys voi vaarantaa yrityksen tai tehtävän elinkelpoisuuden	Toimii joka kerta	Jopa amatöörit voivat toteuttaa helposti	Kaikki	Ei juurikaan havaittavissa
8	Hyökkäys voi aiheuttaa vakavaa vahinkoa	Vaatii muutamien tekijöiden yhteistyötä	Vaatii jotain erityisiä toiminnallisia tai järjestelmän tietoja	Suurin osa	Vaatii merkittäviä resursseja ja tietämystä
5	Hyökkäys voi aiheuttaa merkittävää vahinkoa	Vaatii useiden tekijöiden yhteistyötä	Vaatii kohtuullisen vahvan tunnistuksen ja pääsynvalvonnan läpäisemistä	Monet	Vaatii yksityiskohtaisia ja monimutkaisia tutkimuksia

Taulukko 6: DREAD-mallin arvojen määrittely (jatk.).

3	Hyökkäys voi aiheuttaa kiusaa	Vaatii hyvin vaikeasti hallittavan tapahtumien sarjan	Vaatii hyvin vahvan tunnistuksen ja pääsynvalvonnan läpäisemistä	Muutamia	Vaatii tutkimista, mutta ei liian vaikea havaittavaksi
1	Hyökkäys ei aiheuta vahinkoa	Teoriassa mahdollista, mutta hyvin epätodennäköistä	Vaatii kansallisen tason resursseja.	Ei kukaan	Helppo havaita

Kutakin uhkaa koskeva DREAD-kokonaisriski määritellään laskemalla saaduista arvoista keskiarvo kaavan (2) mukaisesti.

$$RISKI_{DREAD} = (D + R + E + A + D) / 5 \quad (2)$$

Kun kaikkien uhkien riskit on määritelty, uhat tulisi laittaa laskevaan järjestykseen, suurimman riskin omaavat uhat ylimmäiseksi ja matalamman riskin omaavat uhat alimmiksi (Howard ja LeBlanc, 2003).

Small (2004) on kehittänyt DREAD-mallia edelleen niin, että DREAD-arvot eivät ole lineaarisia, vaan esimerkiksi jos vahingon vaikutusmahdollisuuden arvo on 10, niin se merkitsee sitä, että kyseinen riski on huomattavasti yli kymmenen kertaa suurempi kuin jos vahingon vaikutusmahdollisuuden arvo olisi 1. Small (2004) käyttää tekniikastaan nimeä ADDER, joka muodostuu samoista tekijöistä kuin DREAD, mutta uhan aiheuttaman kokonaisriskin laskemiseksi käytetään kaavaa (3).

$$RISKI_{ADDER} = (2^A + 2^D + 2^D + 2^E + 2^R) / 32 \quad (3)$$

Epälineaarisesta laskutavasta johtuen Smallin (2004) käyttämä tekniikka korostaa riskejä, joiden arvo on 10.

Tarkastellaan edelleen aiemmin kuvan 15 hyökkäyspuussa esiteltyä uhkaa, jossa vihamielinen käyttäjä tarkastelee henkilön palkkatietoja. Soveltamalla uhkaan DREAD-mallia voitaisiin saada taulukon 7 esittämät tulokset.

Taulukko 7: STRIDE ja DREAD-mallien avulla luotu uhan kokonaisriskimääritys (Howard ja LeBlanc, 2003).

Ominaisuus	Selitys
Uhan kuvaus	Vihamielinen käyttäjä katselee henkilön palkkatietoja, kun ne ovat matkalla web-palvelimelta selaimelle tai selaimesta web-palvelimelle.
Uhan kohde	Palkkatietojen vastaus
Uhkakategoria	Tietojen paljastaminen **(I)**
Riski	Vahingon vaikutusmahdollisuus: **8** Toistettavuus: **10** Hyödynnettävyys: **7** Vaikutus käyttäjiin: **10** Löydettävyys: **10** Kokonaisriski: ((8+10+7+10+10)/5) = **9**
Kommentti	Yleisimmin kyseinen hyökkäys toteutetaan käyttämällä protokolla-analysaattoria, koska tällä tavoin hyökkäys on helppo toteuttaa. Hyökkäys on passiivinen ja edullinen ajan, työmäärän sekä kustannusten suhteen. Kytkimiä koskeva uhka on tärkeä, koska yleisesti luullaan, että kytkimien avulla toteutettu verkko on turvassa kuuntelulta, mutta niin ei tosiasiassa ole.

3.4 Tapa vastata kuhunkin uhkaan

Kun järjestelmää koskevat uhat on löydetty, tulisi valita tapa, jolla uhkaan pyritään reagoimaan. Howard ja LeBlanc (2003) jaottelevat uhkien lieventämistavat neljään kategoriaan.

Ensimmäinen vaihtoehto on se, että *ongelmalle ei tehdä mitään*. Tämä on harvoin oikea ratkaisu, koska ongelma on sovelluksessa piilevänä ja on mahdollisuus, että ongelmaan törmätään myöhemmin ja se joudutaan korjaamaan. Ei ole myöskään oikein käyttäjiä kohtaan olla tekemättä ongelmalle mitään, koska se voi saada heidät alttiiksi riskille. Jos jostain syystä päätetään olla tekemättä mitään, tulee tarkistaa, voidaanko uhan kohteena oleva ominaisuus ottaa oletuksena pois käytöstä.

Toinen vaihtoehto on *varoittaa käyttäjiä ongelmasta* ja antaa heidän päättää käyttävätkö he uhan aiheuttavaa ominaisuutta vai eivät. Ensimmäisen vaihtoehdon tapaan tämäkin voi olla ongelmallinen, koska monet käyttäjät eivät tiedä, mikä on oikea ratkaisu ja usein ratkaisua vielä vaikeuttaa vaikeaselkoinen ongelmasta kertova teksti. Käyttäjät myös jättävät varoitukset helposti huomioimatta, jos niitä tulee liian usein. Jos päätetään varoittaa käyttäjää ominaisuudesta dokumentissa, ei tule unohtaa, että käyttäjät lukevat dokumentteja harvoin. Tämän vuoksi käyttäjää ei tule koskaan varoittaa pelkästään dokumentissa.

Kolmantena vaihtoehtona on *poistaa ongelma*. Usein kehittäjäryhmillä ei ole aikaa korjata turvallisuusongelmaa, joten he poistavat uhan aiheuttavan ominaisuuden tuotteesta.

Viimeinen ja ilmeisin vaihtoehto vastata uhkaan on *korjata ongelma* teknisesti. Se on myös vaikein, koska se tietää lisää työtä kehittäjille, testaajille ja joissakin tapauksissa myös dokumentoijille.

3.5 Tekniikat joiden avulla uhkia voidaan lieventää

Seuraava vaihe on määritellä, miten löydettyjä uhkia voidaan lieventää. Uhkien lieventämisen tarkoituksena on poistaa ohjelmistossa olevia tietoturvahaavoittuvuuksia. Howardin ja LeBlancin (2003) mukaan tämä on kaksivaiheinen prosessi. Ensin tulee *määritellä tekniikka,* mikä voi auttaa uhan lieventämisessä ja sen jälkeen voidaan *valita sopiva tapa* lieventää uhkaa. Esimerkiksi tunnistus on turvallisuustekniikka ja *Kerberos* on määrätty tapa toteuttaa tunnistus.

Taulukossa 8 on lueteltu joitakin tekniikoita, joita voidaan käyttää STRIDE-mallin avulla luokiteltujen uhkien lieventämiseen.

Taulukko 8: Luettelo uhkien lievennystekniikoista (Howard ja LeBlanc, 2003).

Uhan tyyppi	**Lievennystekniikka**
Identiteetin väärentäminen (S)	Tunnistus Salaisuuksien suojaaminen
Tietojen sekoittaminen (T)	Valtuutus Tiivistefunktio Digitaalinen allekirjoitus Muutoksen estävät protokollat
Torjuminen (R)	Digitaalinen allekirjoitus Seuranta
Tietojen paljastaminen (I)	Valtuutus Yksityisyysperustaiset protokollat Salaus Salaisuuksien suojaaminen
Palvelun esto (D)	Tunnistus Valtuutus Suodatus, rajaaminen, palvelun laatu
Oikeuden nostaminen (E)	Suorittaminen pienimmillä mahdollisilla oikeuksilla

Tunnistuksella tarkoitetaan prosessia, jossa osapuoli varmistaa, että toinen osapuoli on se, joka väittää olevansa. Osapuoli voi olla käyttäjä, jokin suoritettava koodi tai tietokone. Tunnistus tarvitsee todisteen (evidence) valtakirjan (credential) muodossa. Todisteena voidaan käyttää esimerkiksi salasanaa, yksityistä avainta, sormenjälkeä tai muita tunnistustietoja. Kannattaa myös huomioida, että jotkin tunnistustekniikat ovat turvallisempia kuin toiset. McGraw ja Viega (2002) muistuttavat, että salasanaan pohjautuvat tunnistusmenetelmät eivät ole kovin vahvoja, vaikka ovatkin yleisessä käytössä. Tunnistuksen yhteydessä olisi hyvä käyttää

alakohdassa 4.1.2 tarkasteltavaa perusteellisten suojausten periaatetta yhdistämällä useita tunnistusmenetelmiä.

Kun yhteydenottajan identiteetti on määritelty tunnistuksen avulla, hän yleensä haluaa käyttää jotain resurssia, kuten tiedostoa tai tulostinta. *Valtuutus* tarkoittaa tarkistuksen avulla tehtävää määrittelyä siitä, onko tunnistetulla käyttäjällä oikeus käyttää pyytämäänsä resurssia. Joillakin käyttäjillä voi olla laajemmat oikeudet resursseihin kuin toisilla.

Useat verkkoprotokollat tukevat *muutosten estoa* ja *tietojen yksityisyyttä*. Muutoksen esto viittaa ominaisuuteen suojata tiedot tuhoamiselta tai muutokselta, tapahtui se sitten haitantekomielessä tai vahingossa. Verkkoprotokollan tapauksessa yksityisyys tarkoittaa sitä, että käyttäjällä on mahdollisuus tarkastella tietoja vain, jos tiedot on tarkoitettu kyseiselle käyttäjälle.

Paras tapa *suojata salaiset tiedot* on olla tallentamatta niitä. Sovelluksen kaatuessa, hyökkääjä ei pääse käsiksi salaisiin tietoihin, koska niitä ei ole tallennettu. Jos on välttämätöntä tallentaa salaisia tietoja, tiedot tulee turvata niin hyvin kuin mahdollista.

Salausmenetelmien avulla pyritään varmistamaan tietojen luottamuksellisuus, eheys ja kiistämättömyys. Käytetyn salauksen tulisi olla sellainen, jonka murtaminen kohtuullisessa ajassa ja kohtuullisin resurssein ei olisi mahdollista.

Tiivistefunktion (hash- tai digest-funktio) tarkoituksena on laskea pitkästä syötteestä lyhyt, vakiopituinen tiiviste, jota voidaaan käyttää tarkastettaessa tiedon alkuperäisyyttä. Tarkastaminen tapahtuu laskemalla saaduista tiedosta uusi tiivistearvo ja vertaamalla sitä tiedon mukana tulleeseen tiivistearvoon.

Digitaalisessa allekirjoituksessa lähettäjä laskee lähetettävästä viestistä tiivistearvon, jonka hän salaa yksityisellä avaimellaan. Vastaanottaja avaa salatun tiivistearvon lähettäjän julkisella avaimella, laskee itse viestistä tiivistearvon ja vertaa sitä lähettäjän julkisella avaimella avaamaansa tiivistearvoon. Jos tiivistearvot ovat samat, on viesti todistettavasti allekirjoitettu lähettäjän yksityisellä avaimella eikä kukaan ole muuttanut viestiä matkalla.

Seurannan tavoite on kerätä tietoja onnistuneista ja epäonnistuneista objektien käytöstä, oikeuksien käytöstä ja muista tärkeistä turvallisuuteen liittyvistä toiminnoista ja kirjoittaa ne pysyvään varastoon myöhempää analyysiä varten. Seuranta ei ole varsinainen lievennnystekniikka, mutta tieto seurannasta voi jossain tapauksissa vähentää mahdollisia hyökkäyksiä (McGraw ja Viega, 2002).

Suodattaminen tarkoittaa vastaanotettujen tietojen tarkistamista ja joko niiden hyväksymistä tai hylkäämistä. Monet IP-tason palvelunestohyökkäyksistä on mahdollista lieventää paketteja suodattavan palomuurin avulla. *Rajaaminen* takoittaa järjestelmään tulevien pyyntöjen rajoittamista. Esimerkiksi on mahdollista sallia vain pieni määrä anonyymeja pyyntöjä, mutta sallia enemmän pyyntöjä tunnistetuilta käyttäjiltä. Tämä voi osaltaan parantaa turvallisuutta, koska hyökkääjä ei ehkä yritä hyökkäystä, jos hänen pitää ensin kirjautua käyttäjäksi. *Palvelun laatu* on joukko komponentteja, joiden avulla on mahdollista tarjota erikoiskäsittelyä määrätyn tyyppiselle liikenteelle.

Ohjelmat tulee suorittaa aina *pienimmillä mahdollisilla oikeuksilla*. Pienimmän mahdollisen oikeuden periaatetta käsitellään tarkemmin alakohdassa 4.1.4.

Kuvassa 15 ja taulukossa 7 käsitellyn uhan tapauksessa, jossa vihamielinen käyttäjä katselee henkilön palkkatietoja, kun ne ovat matkalla web-palvelimelta selaimelle tai selaimesta web-palvelimelle, on kysessä tietojen paljastaminen. Tietojen paljastamisuhkaa voidaan lieventää käyttämällä yksityisyyttä varjelevaa tekniikkaa, kuten esimerkiksi salausta (esimerkiksi SSL/TLS) asiakkaan ja web-palvelimen välisessä tiedonsiirrossa ja estää näin hyökkääjää tutkimasta tietoja tiedonsiirron aikana.

4. Periaatteet ja tarkastuslistat

Tässä luvussa esitellään asioita, joihin tulisi kiinnittää huomiota, sovellusta suunniteltaessa tai toteutettaessa. Turvallisuuden ei tulisi olla jotakin, joka voidaan eristää määrätystä osasta koodia. Suorityskyvyn, skaalautuvuuden, hallittavuuden ja koodin luettavuuden tapaan turvallisuustietoisuus on alue, joka jokaisen sovelluksen suunnittelijan, ohjelmoijan ja testaajan tulee hallita.

4.1. Periaatteita ja käytäntöjä tietoturvan parantamiseksi

Tässä kohdassa esiteltävien ohjaavien toimintaperiaatteiden päämäärä on tunnistaa ja tuoda esiin tärkeimmät tavoitteet, joita tulisi noudattaa suunniteltaessa ja toteutettaessa ohjelmistoa. Seuraamalla toimintaperiaatteita on mahdollista välttää monet yleisistä tietoturvaongelmista. Ohjaavien toimintaperiaatteiden tarkoituksena on parantaa ohjelmistojen turvallisuutta myös kohdatessa tuntemattomia hyökkäyksiä, joita vastaan suojautuminen on paljon suurempi haaste, kuin suojautua paljon käytetyiltä rutiinihyökkäyksiltä.

4.1.1 Heikoimman lenkin turvaaminen

McGrawn ja Viegan (2002) mukaan tietoturva-asiantuntijat esittävät usein, että tietoturva on tapahtumaketju, jolloin myös ohjelmiston tietoturva on yhtä vahva kuin ketjun heikoin lenkki. Hyökkääjien pyrkimyksenä on hyökätä järjestelmän heikoimpia kohtia vastaan, koska ne ovat yleensä helpoimpia murtaa.

Järjestelmän heikoimman komponentin tunnistaminen perustuu hyvään riskianalyysiin. Tietoturvaresurssit tulisi jakaa uhkien perusteella niin, että ensin keskityttäisiin pääongelmiin ja siirryttäisiin seuraaviin vakavuusasteen perusteella. Tätä strategiaa on mahdolista toteuttaa loputtomiin, koska täydellistä tietoturvaa ei voida koskaan saavuttaa. On hyväksyttävää lopettaa riskien tarkastelu, kun kaikki komponentit vaikuttavat olevan tietoturvan perusteella hyväksyttävän rajan sisällä (McGraw ja Viega, 2002).

4.1.2 Perusteellisten suojauksien käyttäminen

McGrawn ja Viegan (2002) mukaan idea perusteellisten suojausten toteuttamiseksi on hallita riskejä monenlaisten puolustusstrategioiden avulla niin, että puolustuksen yhden tason muuttuessa riittämättömäksi puolustuksen toisen tason toivotaan pystyvän estämään murtautumisen. Howard ja LeBlanc (2004) esittävät, että jonkin puolustusmekanismin murtuminen ei saa olla syy luovuttaa puolustautumista hyökkäystä vastaan. Ohjelmistojen ei tulisi myöskään luottaa muiden järjestelmien tarjoamaan suojeluun, vaan ohjelmiston tulee puolustautua itse mahdollisia hyökkäyksiä vastaan.

Perusteellisten suojausten toimintaperiaate voi vaikuttaa olevan ristiriidassa heikoimman lenkin turvaamisen kanssa, koska tässä periaatteessa sanotaan, että puolustus kokonaisuutena voi olla vahvempi kuin sen heikoin lenkki. Kuitenkaan kyseessä ei ole ristiriita, koska on mahdollista, että useiden komponenttien yhteenlaskettu suojaus on suurempi, kuin pelkästään yhden komponentin tarjoama suojaus.

4.1.3 Epäonnistuminen turvallisesti

Epäonnistuminen voi tapahtua turvallisesti tai turvattomasti. Epäonnistuminen turvallisesti tarkoittaa, että sovellus ei ole paljastanut mitään sellaista tietoa, jota se ei normaalitilanteessakaan paljasta. Epäonnistuessaan turvattomasti sovellus paljastaa enemmän tietoa, kuin mitä sen pitäisi, tai sovelluksen tiedot vioittuvat. Jos hyökkääjä pystyy vioittamaan koodia, hän voi ohittaa turvamekanismin. Turvamallissa tulisi huomioida myös käyttäjän syöttämien tietojen suodattaminen niin, että mahdolliset vihamieliset syötteet hylätään (McGraw ja Viega, 2002).

McGraw ja Viega (2002) toteavat, että jokaisella tarpeeksi monimutkaisella järjestelmällä on häiriötilanteita. Jos toimintahäiriö saa järjestelmän toimimaan epävarmasti, niin tällaisissa tapauksissa järjestelmään on mahdollista hyökätä aiheuttamalla järjestelmälle oikeanlainen toimintahäiriö. Vikakäytäntöjen avulla pyritään varautumaan järjestelmän mahdolliseen toimintahäiriöön. Tällä tavoin toimintahäiriöön liittyvät tietoturvaongelmat ovat vältettävissä.

4.1.4 Pienimmän mahdollisen oikeuden periaatteen käyttäminen

Howard ja LeBlanc (2003) sekä McGraw ja Viega (2002) esittävät, että sovellukset tulee suorittaa pienimmillä mahdollisilla oikeuksilla, joilla ne saadaan suoritetuksi ja oikeudet tulee myöntää mahdollisimman lyhyeksi ajaksi. Jos ohjelmakoodista löydetään turvallisuusheikkous ja hyökkääjä pystyy syöttämään ohjelmakoodia järjestelmään, ajetaan hyökkääjän syöttämä koodi samoilla oikeuksilla kuin varsinainen prosessi.

Kun annetaan oikeuksia päästä käsiksi joihinkin järjestelmän osiin, niin siihen liittyy aina myös riskejä, että oikeuksia käytetään väärin. Neumann (2004) esittää, että eri käyttäjille tulisi myöntää oikeuksia vain sen mukaan mitä kukin käyttäjä tarvitsee pystyäkseen suorittamaan vaaditut toimenpiteet. Tällaisessa tilanteessa tulisi myös kiinnittää huomiota käyttäjien tunnistukseen, että kukaan ei pystyisi käyttämään toiselle käyttäjälle myönnettyjä vahvempia oikeuksia.

McGrawn ja Viegan (2002) mukaan ohjelmoinnissa voi tulla houkutusta antaa liikaa oikeuksia. Esimerkiksi jos johonkin objektiin tarvitaan lukuoikeus, mutta ohjelmoija ajattelee, että hän saattaa joskus tulevaisuudessa tarvita myös kirjoitusoikeuksia kyseiseen objektiin, niin hän antaa objektiin myös kirjoitusoikeudet. Tämän kaltaiset toiminnat saattavat johtaa myös väärinkäytöksiin, jos hyökkääjä pystyy hyödyntämään ohjelmoituja takaportteja.

4.1.5 Osiin jakaminen

Osiin jakamisen perusidea on minimoida järjestelmälle tehtävän vahingon määrä jakamalla järjestelmä niin pieniin yksiköihin kuin mahdollista. Osiin jakamista tulee kuitenkin käyttää kohtuudella, sillä eristämällä jokaisen toiminnallisuuden tulee järjestelmästä vaikeasti hallittava (McGraw ja Viega, 2002). Osiin jakaminen on myös uhkamallinnuksen väline kohdan 3.1 mukaisesti.

4.1.6 Hyökkäyskohteiden vähentäminen

McGraw ja Viega (2002) esittävät monimutkaisuuden lisäävän tietoturvaongelmien riskiä. Tämän takia ohjelmiston suunnitelun ja toteutuksen tulisi olla mahdollisimman suoraviivaista. Monimutkainen suunnittelu ei ole koskaan helppoa ymmärtää ja siksi se sisältää hyvin todennäköisesti pieniä ongelmia, joita ei löydetä suunnitteluvaiheen aikana. Monimutkainen ohjelmakoodi on myös vaikeammin hallittavaa ja se sisältää yleensä enemmän virheitä kuin yksinkertainen koodi. Hyvälaatuisia uudelleenkäytettäviä komponentteja kannattaa myös käyttää aina, kun se vain on mahdollista. Mitä onnistuneemmin tiettyä komponenttia on nähty käytettävän, sitä suuremmalla syyllä komponenttia ei tulisi kirjoittaa uudelleen.

Mahdollisimman yksinkertaisen, mutta kuitenkin turvallisuusvaatimukset täyttävän ohjelmiston valmistaminen ei ole aina helppoa. Yksinkertaisuutta voidaan usein tehostaa ohjaamalla kaikki tietoturvan kannalta kriittiset operaatiot tiettyjen järjestelmän kuristuspisteiden (choke point) kautta, koska on paljon helpompi valvoa käyttäjän käyttäytymistä ja syötteitä, jos ne on pakotettu muutamiin pieniin kanaviin (McGraw ja Viega, 2002). Muutaman kuristuspisteen kautta tapahtuvaa liikennettä on myös helpompi tallentaa mahdollisen hyökkäyksen jäljityksen apuna käytettäviin lokitiedostoihin kuin usean eri reitin kautta tapahtuvaa toimintaa (Neumann, 2004).

Yksi vähemmän ilmeinen, mutta tärkeä seikka yksinkertaisuudessa on myös käytettävyys. Käytettävyys vaikuttaa niin ihmisiin, jotka käyttävät ohjelmistoa, kuin myös henkilöihin, jotka ylläpitävät koodia (McGraw ja Viega, 2002). Neumannin (2004) mukaan vaikeakäyttöiset turvallisuusominaisuudet vähentävät niiden käyttöä.

4.1.7 Yksityisyyden edistäminen

McGrawn ja Viegan (2002) mukaan monet käyttäjät pitävät yksityisyyttä osana turvallisuutta. Ohjelmistojen suunnittelussa ja toteutuksessa tulisi aina tähdätä mahdollisimman hyvään käyttäjän yksityisyyteen. Yleensä käytettävyys ja salassapito eivät kulje käsi kädessä, sillä esimerkiksi käytettävyyden kannalta ei ole hyvä asia, että monet järjestelmät unohtavat käyt-

täjän salasanan heti käytön jälkeen. Salassapidon kannalta tämän kaltainen asia on kuitenkin välttämätöntä.

Käyttäjän yksityisyys ei ole ainoa yksityisyyden tyyppi. Hyökkääjät pyrkivät käynnistämään hyökkäyksiä kohdekoneesta helposti kerättävän tiedon perusteella. Kohdekoneessa toimivat palvelut antavat usein tietoa itsestään, mikä voi auttaa hyökkääjää keksimään hyökkäystapoja. Esimerkiksi TELNET-palvelu tarjoaa käyttöjärjestelmän nimen ja version kuvan 17 mukaisesti.

```
$ telnet cs.joensuu.fi
Trying 193.167.42.127...
Connected to cs.joensuu.fi.
Excape character is '^]'.

SunOS 5.9
```

Kuva 17: Telnet-yhteyden tarjoamat tiedot palvelimesta.

Järjestelmän tietojen esille jättäminen voi helpottaa mahdollisia hyökkääjiä. Tällaisissa tapauksissa hyökkääjän johtaminen harhaan väärän tiedon avulla voi heikentää hyökkäyksen onnistumista (McGraw ja Viega, 2002).

4.1.8 Salaisuuksien piilottamisen vaikeus

Tietoturva on usein salaisuuksien säilyttämistä. Käyttäjät eivät halua, että toisilla henkilöillä on mahdollisuus katsella heidän henkilökohtaisia tietojaan. Myös huippusalaiset algoritmit täytyy suojata kilpailijoilta. McGrawn ja Viegan (2002) mukaan tämän kaltaiset vaatimukset ovat melkein aina tietoturvavaatimuksista tärkeimpiä, mutta näiden tavoitteiden saavuttaminen osoittautuu paljon vaikeammaksi, mitä yleensä ajatellaan.

McGrawn ja Viegan (2002) mukaan useat tutkimukset ovat osoittaneet, että yleisin uhka yrityksille ovat sisältäpäin tulevat hyökkäykset, joissa tyytymätön työntekijä väärinkäyttää omia oikeuksiaan. Leachin (2003) mukaan yrityksen sisäinen uhka aiheutuu lähinnä työntekijöiden heikosta tietoturvakäyttäytymisestä. Sisäinen tietoturvauhka kattaa työntekijöiden tekemien

virheiden ja laiminlyöntien lisäksi myös huolimattomuuden sekä harkitut teot yritystä vastaan. Sisäiset tietoturvauhat johtuvat pääasiassa työntekijöiden *tietoturvallisen ajattelukyvyn puutteesta*, joka ilmenee esimerkiksi epäilyttävien liitetiedostojen avaamisena, *tietoturvamenettelytapojen laiminlyönneistä*, *väärästä turvallisuuden tunteesta*, esimerkiksi tietokoneen lukitsematta jättäminen poistuttaessa sen luota, *harkituista laiminlyönneistä,* kuten salaisten asiakirjojen lähettäminen sähköpostitse ilman salausta, ja *harkituista hyökkäyksistä* esimerkiksi kertomalla tietoturvariskeistä ulkopuolisille.

Ihmisten tietoturvakäyttäytymiseen vaikuttavat tekijät voidaan jakaa kahteen pääryhmään. Ensimmäinen ryhmä sisältää käyttäjän käsityksen siitä, millaisia toimintatapoja yritys odottaa käyttäjiltä. Tähän seikkaan voidaan vaikuttaa *käyttäjälle kerrottujen asioiden* avulla esimerkiksi käyttämällä *tietoturvaopasta*, joka sisältää yrityksen tietoturvastandardit ja menettelytavat. Vaikutusta on myös *käyttäjien näkemillä esimerkeillä* ja *käyttäjien tekemillä päätöksillä*. Käyttäjien näkemät esimerkit vaikuttavat myös käyttäjien myöhemmin tekemiin tietoturvallisuutta koskeviin päätöksiin. Etenkin jos tietoturvaoppaan ja todellisten toimintatapojen välillä on ristiriitaisuuksia, niin uudet käyttäjät omaksuvat helpommin toimintatavat, joita he näkevät käytännössä, verrattuna toimintatapoihin, joita heille kerrotaan. Käyttäjien tekemistä tietoturvapäätöksistä olisi myös hyvä antaa palautetta, koska palautteen puuttuessa käyttäjä yleensä hyväksyy tekemänsä päätöksen ja on valmis käyttämään sitä uudelleen, vaikka tehty päätös ei oliskaan tietoturvan kannalta hyvä (Leach, 2003).

Toinen ryhmä sisältää tekijöitä, jotka vaikuttavat käyttäjien haluun hillitä toimintatapojaan niin, että ne pysyvät hyväksyttyjen normien sisällä. Tähän ryhmään kuuluvat käyttäjien *henkilökohtaiset arvot*. Jos yksilön ja yrityksen arvojen välillä on ristiriitoja, se voi aiheuttaa jännitteitä ja samalla käyttäjän epälojaaliutta yritystä kohtaan. Toinen ryhmään kuuluva seikka on *käyttäjän velvollisuudentunne*. Yleisesti työntekijä sitoutuu työskentelemään tunnollisesti sekä mukautuu yrityksen käyttäytymismalliin, joiden vastineena yritys kohtelee työntekijöitä hyvin. Jos työntekijä tuntee tulleensa kohdellun huonosti, niin tämän seurauksena hän ei myöskään välttämättä aina ota yrityksen etua huomioon. Joissain tapauksissa työntekijä saattaa myös haluta kostaa yrityksessä saamansa huonon kohtelun ja tämän seurauksena työntekijästä tulee tietoturvauhka. Kolmas ryhmään kuuluva asia ovat *mukautusmivaikeudet*, esimerkiksi jos tietoturvan kontrollointi on vaikeaa tai toiminnallisesti rasittavaa ja se tarjoaa

vain vähän näkyvää hyötyä, niin käyttäjät saavat luonnollisen yllykkeen kiertää kontrollointia. Vaikka käyttäjät tiedostaisivatkin, että tietoturvan kontrollointiin on hyvät syyt, niin heillä on hyvin vähän suvaitsevaisuutta kontrolloinnille, joka ei ole tehokasta (Leach, 2003).

Näistä edellä esitetyistä tekijöistä käyttäjien näkemät esimerkit, käyttäjien tekemät päätökset ja käyttäjän velvollisuudentunne ovat avainasemassa kehitettäessa tietoturvakäyttäytymistä ja vähennettäessä sisäistä tietoturvariskiä. Leach (2003) esittää artikkelissaan, että yrityksen tulisikin keskittyä käyttäjiin suunnatussa tietoturvastrategiassaan juuri näiden tekijöiden kehittämiseen, sillä vaikutusmahdollisuudet käyttäjän henkilökohtaisiin arvoihin ja ajatuksiin ovat hyvin vähäiset.

4.1.9 Luottamuksen rajoittaminen

McGrawn ja Viegan (2002) mukaan on hyvin yleistä, että salaisuuksia piilotetaan ohjelmakoodiin, olettaen että salaisuudet ovat turvassa. Ongelmana tässä on se, että lahjakas loppukäyttäjä voi väärinkäyttää asiaa ja saada selville ohjelmakoodin kaikki salaisuudet. Tämän takia asiakkaiden ja palvelimien välistä luottamusta tulisi pyrkiä rajoittamaan, koska on mahdollista, että palvelimeen tai käyttäjän koneeseen on murtauduttu.

Joskus on viisasta olla luottamatta myös itseensä (McGraw ja Viega, 2002) Tämän vuoksi olisi hyvä tietyin väliajoin pyytää myös ulkopuolisten arviota siitä, mitä itse on tekemässä. Yksi seikka on myös muistaa, että luottamus on usein periytyvää. Tästä syystä luotettujen ohjelmien ei tulisi koskaan käynnistää ohjelmia, joilla ei ole luottamustamme.

Howardin ja LeBlancin (2004) mukaan myöskään ulkoisiin järjestelmiin ei tulisi luottaa. Ulkoisten järjestelmien pitäminen turvattomana liittyy perusteellisen suojauksen käyttämiseen. Tämä on tärkeä muistaa myös, kun hyväksytään käyttäjän syöttämiä tietoja.

4.1.10 Yhteisön resurssien hyödyntäminen

Vaikka ei tule luottaa ulkoisiin järjestelmiin, niin siitä huolimatta joissain tilanteissa on järkevää käyttää sovelluksessa ulkopuolisten valmistamia ominaisuuksia. Esimerkiksi salauksen tapauksessa ei tule luottaa mihinkään algoritmiin, joka ei ole yleisessä tiedossa ja jota ei ole laajasti tutkittu (McGraw ja Viega, 2002).

Monet kehittäjät kirjoittavat omia salausalgoritmeja, joskus luottaen siihen seikaan, että vaikka ne olisivat heikkoja, niin niiden vaikeaselkoisuus auttaa. Tämä on kuitenkin harhakuvitelmaa. Kuten Howard ja LeBlanc (2004) toteavat, koskaan ei kannata luottaa tuntemattomaan turvallisuuteen. Ohjelmoijan tulee olettaa aina, että hyökkääjä on saanut käsiinsä kaiken ohjelmistoissa käytetyn lähdekoodin ja suunnitelmat ja tällä tavoin hyökkääjä tietää järjestelmästä saman kuin ohjelmoijakin.

Caloyannidesin & al. (2001) mukaan ohjelmistojen tietoturvallisuuteen voidaan vaikuttaa myös tekemällä lähdekoodista vapaasti levitettävää. Avoin lähdekoodi lisää koodin ulkoista katselmointia ja tämä parantaa ohjelmiston tietoturvaheikkouksien löytymisnopeutta. Toisaalta avoin lähdekoodi on myös hyökkääjien saatavissa ja hyökkääjä voi helposti etsiä koodista sopivia haavoittuvuuksia. Vaikka lähdekoodi ei olisi vapaasti saatavilla niin hyvin monissa tapauksissa jo käännetystä ohjelmistosta voidaan oikeiden työkalujen avulla tutkia myös lähdekoodia. Perusteltaessa suljetun lähdekoodin etuja verrattuna avoimeen lähdekoodiin ei kannata luottaa tuntemattomaan turvallisuuteen.

4.1.11 Tuotteen turvallisuustavoitteiden määrittely

Tuotteen turvallisuustavoitteiden määrittely tulee tehdä niin, että se vaatii mahdollisimman vähän tuotekohtaista tietämystä. Howardin ja LeBlancin (2003) mukaan on välttämätöntä, että jokainen projektissa mukana oleva henkilö ymmärtää vaatimukset tietoturvan osalta.

Turvallisuustavoitteiden määrittelyssä tulee laatia dokumentti, joka vastaa seuraaviin kysymyksiin (Howard ja LeBlanc, 2004):

- Mikä ovat sovelluksen kohderyhmä?
- Mitä turvallisuus merkitsee kohderyhmälle? Eroaako se kohderyhmän yksittäisten henkilöiden välillä? Onko eri asiakkailla erilaisia turvallisuusvaatimuksia?
- Missä sovellusta käytetään? Internetissä? Palomuurin takana?
- Mitä yritetään suojella?
- Mitä seuraamuksia käyttäjille aiheutuu, jos suojaavat asiat paljastuvat?
- Kuka hallinnoi sovellusta? Käyttäjä vai yrityksen IT-osasto?
- Mitkä ovat tuotteen tietoliikennevaatimukset? Onko tuote organisaation sisäsinen, ulkoinen vai molempia?
- Mitä sellaisia turvajärjestelmiä käyttöjärjestelmä tai ajoympäristö tarjoavat, joita on mahdollista hyödyntää?
- Kuinka käyttäjää tule suojata hänen omilta toimiltaan?

Suurin osa näistä tiedoista voidaan kerätä uhkamallista (Howard ja LeBlanc, 2004).

4.1.12 Virheistä oppiminen

Turvallisuusongelmien löytyessä, kannattaa etsiä vastauksia seuraaviin kysymyksiin (Howard ja LeBlanc, 2003):

- Miten virhe tapahtui?
- Onko sama virhe toistettu koodin muissa osissa?
- Miten olisimme voineet estää virheen tapahtumisen?
- Miten varmistamme, että tämän tyyppisiä virheitä ei tapahdu tulevaisuudessa?
- Täytyykö koulutusta tai analysointityökaluja päivittää?

Jokaista virhettä kannattaa käsitellä oppimismahdollisuutena. Howardin ja LeBlancin (2003) mukaan tietoturvavirheen löytyessä jokaisesta löytyneestä virheestä olisi hyvä muodostaa raportti, joka sisältäisi ainakin seuraavat tiedot:

- tuotteen nimi

- tuotteen versionumero
- yhteyshenkilön nimi
- virheen tietokantanumero
- haavoittuvuuden kuvaus
- haavoittuuvuuden merkitys
- ilmeneekö asia ohjelmiston perusasennuksessa
- kuvaus siitä, mitä suunnittelijat, kehittäjät tai testaajat olisivat voineet tehdä estääkseen tämän ongelman
- korjauksen kuvaus, mukaanluettuna erot alkuperäisessä ja korjatussa koodissa.

Jos tietoturvavirheestä ei muodosteta raporttia ja löytyneitä virheitä ei hyödynnetä oppimistarkoituksessa, niin on todennäköistä, että samoja virheitä toistetaan tulevaisuudessa.

4.2. Tarkastuslistat

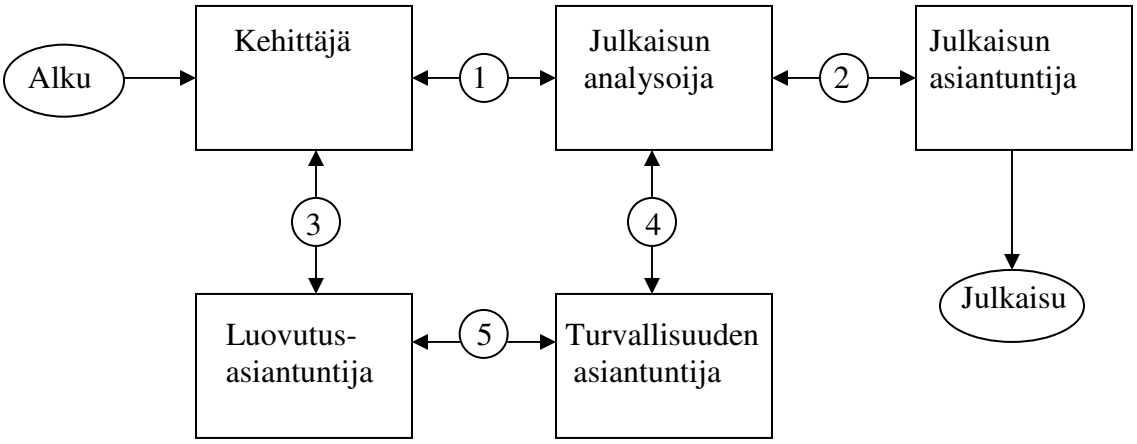
Gilliamin & al. (2003) mukaan tarkastuslistoja voidaan hyödyntää ohjelmistojen tietoturvan parantamiseksi kaikissa ohjelmiston elinkaaren vaiheissa. Tärkeintä tarkastuslistojen läpikäynti on kuitenkin ohjelmiston kehitysvaiheessa, jossa tarkoituksena on tarkastaa ohjelmakoodia niin, että se täyttää tarkastuslistojen vaatimukset. Tarkastuslistojen luomiseksi täytyy suorittaa uhkamallinnus tai vaihtoehtoisesti jokin muu tietoturvariskianalyysi, jossa tunnistetaan ohjelmistoa koskevat tietoturva-asiat ja -vaatimukset. Riskianalyysin ja vaatimusmäärittelyn tulisi ohjata loput elinkaaresta jäljitettävillä ja varmennetuilla tietoturvavaatimuksilla. Ohjelmistolla tulisi myös olla riskitason luokitus, jonka luokitteluun voidaan käyttää esimerkiksi luvussa 3 esiteltyä DREAD-mallia. Tämä luokitus mahdollistaa käyttäjien tietoisuuden ohjelmiston kokonaisturvallisuudesta.

Taulukossa 9 on kuvattu joitakin kriittisiä alueita, joita voidaan hyödyntää tietoturvatarkastuslistojen luomisessa. Listaa voidaan laajentaa, muuttaa tai korvata.

Taulukko 9: Tarkastuslistan luomisessa huomioitavia seikkoja (Gilliam & al., 2003).

1	Läpikäynnin esittely, tietoturvakatselmoinnin tulos tai virallinen tietoturvakatselmointi jokaisessa ohjelmiston selinkaaren vaiheessa.
2	Kohderyhmän määrittely, sekä yhteisten tietoturvavaatimuksien esittely ja määrittely jokaiselle kohderyhmälle.
3	Yhteyksien ja ohjelmistotuotteen potentiaalisen käytön sekä tarpeellisten tietoturvavaatimusten määrittely yhdessä käyttöympäristön kanssa.
4	Ennen järjestelmän arkkitehtuurista suunnittelua tulee ohjelmoijilla, kehittäjillä, katselmoijilla ja testaajilla olla saatavilla ohjelmointikieliin ja käyttöjärjestelimiin liittyvät haavoittuvuudet ja potentiaaliset riskit.
5	Tarkistus ohjelmistojen tietoturvariskien lähteiden varalta, kuten ristiriidat vaatimusmäärittelyssä ja suunnittelussa ja uudelleenkäytettävät ohjelmat.
6	Turvattomien rutiinien käytön välttäminen.
7	Ohjelmistossa käytettyjen väliohjelmistojen turvallisuuden tarkastus.
8	Toteutuskohtaisten haavoittuvuuksien, kuten kilpailutilanteiden tarkastus.
9	Takaporttien estäminen.
10	Käyttäjän syötteitä lukevien lähdekoodin kohtien tunnistaminen.
11	Turvattomista lähteistä syötteitä saavien lähdekoodin kohtien tunnistaminen.
12	Ohjelman mahdollisten syötteiden lähteiden tarkastus.
13	Varmennettujen yhteyksien tarkastus.
14	Ohjelmiston monimutkaisuuden tarkastus ja vaihtoehtojen etsiminen monimutkaisuuden vähentämiseksi.
15	Vakioasetuksena ”korkein turvallisuus” sekä hyväksyntä poikkeuksille.
16	Tietoturvariskien luokitusjärjestelmän muodostaminen ja ohjelmistotuotteen luokittelu.

Ennen ohjelmiston julkaisua tulisi suorittaa arviointi/hyväksymisprosessi. Kuva 18 esittää mahdollista julkaisuprosessia.



Kuva 18: Ohjelmiston julkaisuprosessi (Gilliam & al., 2003).

Kuvassa 18 esitettyjä eri toimintoja on mahdollista suorittaa myös saman roolin toimesta. Tarkastellaan tarkemmin kuvausta kustakin rooleista ja eri roolien tehtävistä (Gilliam & al., 2003):

Kehittäjän (developer) tehtävänä on luoda tuote joka julkaistaan. Julkaistavat tuotteet lähetetään *julkaisun analysoijalle* (release analyst), joka analysoi kehittäjän luoman tuotteen ja määrittelee vastaako tuote julkaisukriteereitä. Kaikella tuotteessa olevalla arkaluontoisella tiedolla täytyy olla luovutuslupa (waiver) tai muuten tuote lähetetään takaisin kehittäjälle korjattavaksi.

Julkaisun asiantuntijan (release authority) tehtävä on ottaa vastaan julkaistava tuote, luovutusluvat ja ohjelmiston tietoturvallisuutta käsittelevä tarkastuslista. Jos tuote vastaa julkaisukriteereitä, niin julkaisun asiantuntija hyväksyy tuotteen julkaisun. *Luovutusasiantuntija* (waiver authority) jakaa luovutusluvat kaikelle julkaistavassa tuotteeseen jäljelle jäävälle arkaluontoiselle tiedolle. Luovutuslupien asiantuntija voi tarvita neuvoja t*ietoturva-asiantuntijalta* (security authority), jonka tehtävänä on tehdä päätös, voidaanko jokin osa ohjelmiston arkaluontoisista tiedoista poistaa.

Taulukko 10 sisältää joitakin tarkastuslistan aiheita ja kysymyksiä, jotka tulisi huomioida ohjelmiston julkaisuvaiheessa.

Taulukko 10: Julkaisussa käytettävän tarkastuslistan osa (Gilliam & al., 2003).

1.0	Sisältääkö ohjelmisto IP-osoitteitta ja aliverkon alueita? 1.1 Jos kyllä, niin ovatko nämä IP-osoitteet arkaluontoisia? 1.2 Voidaanko näiden osoitteiden avulla hankkia luottamuksellista tietoa?
2.0	Sisältääkö ohjelmisto isäntäkoneiden nimiä? 2.1 Jos kyllä, niin ovatko nimet arkaluontoisia? 2.2 Aiheuttaako nimien julkaiseminen riskin?
3.0	Onko ohjelmistossa asetuksia, jotka voidaan käyttää hyväksi? 3.1 Jos kyllä, niin voidaanko näitä asetuksia muokata tai poistaa? 3.2 Jos asetuksia ei voida muokata, eikä poistaa, niin aiheuttaa se mahdollisen riskin? 3.3 Jos asetuksia voidaan muokata tai poistaa, niin aiheuttaako se mahdollisen riskin?
4.0	Onko ohjelmistossa ei-arkaluontoisia tietoja, joiden avulla voidaan paljastaa salaisuuksia? 4.1 Jos on, niin voidaanko ei-arkaluontoisen tiedon avulla paljastaa arkaluontoisia tietoja? 4.2 Voidaanko ei-arkaluontoista tietoa muuttaa niin, että se aiheuttaa riskin?
5.0	Onko ohjelmistossa materiaalia, joka voi ilmiantaa yrityksen tietoja, kuten asiakaslista? 5.1 Jos on, niin onko asiakaslista suojattu tietoturvapolitiikan mukaisesti? 5.2 Aiheuttaako asiakaslistan julkaisu riskin? 5.2 Aiheuttako asiakaslistan julkaisu harmia asiakkaille?
6.0	Ovatko mitkään tiedot rajoitettuja? 6.1 Jos ovat, niin valvotaanko tietoja tietoturvamekanismin avulla? 6.2 Jos ovat, niin vastaavatko tietojen siirtämisessä käytetyt tietoturvarajoitukset rajoitettujen tietojen tietoturvavaatimuksia? 6.3 Siirretäänkö rajoitettuja tietoja avoimen verkon välityksellä? 6.3.1 Ovatko rajoitetut tiedot salattu ennen siirtoa?

5. Yhteenveto

Haavoittuvuuksien poistamiseksi tietoturvan tulisi olla osa koko ohjelmiston elinkaarta. Keskeinen vaihe ohjelmiston elinkaaren aikana suoritetuista tietoturvaa parantavista tomenpiteistä on ohjelmiston kehitysvaiheessa suoritettava uhkamallinnus. Uhkamallinnuksen ensimmäinen vaihe on löytää sovellusta koskevat uhat. Tämän jälkeen, löytyneet uhat tulisi kategorisoida käyttäen STRIDE-mallia sekä järjestää uhat DREAD-mallin avulla määriteltyjen arvojen perusteella. Lopuksi tulisi valita tapa, jolla kuhunkin uhkaan vastataan sekä lievennystekniikka, jolla kutakin uhkaa pyritään vähentämään.

Ohjelmistojen kehitysvaiheessa tulisi myös järjestää tietoturvasysäyksiä, joiden avulla voidaan parantaa kehittäjien mahdollisuuksia tehdä turvallisempia ohjelmistoja, koska tietoturvasysäyksen tavoitteena on opettaa välttämään yleisimpiä tietoturvaa koskevia virheitä ja antaa esimerkkejä siitä, kuinka turvallisempia ohjelmistoja voidaan tehdä. Koulutuksen vaikutusta tietoturvaan ei tulisi myöskään vähätellä, vaan koko henkilöstölle tulisi tietyin väliajoin järjestää tietoturvakoulutusta. Tällä tavoin olisi mahdollista vastata nopeasti uusiin tietoturvauhkiin.

Ohjelmistojen toteutukseen liittyy olennaisena osana myös testaus, joskin tietoturvaominaisuuksien testaaminen voi joissakin tapauksissa olla hankalaa, aikaa vievää ja se voi vaatia uudenlaisten testausmenettelytapojen luomista. Usein testaaminen perustuu siihen, että ohjelman ominaisuuksia testataan antamalla ohjelmalle syötteenä oikeellisia arvoja ja tällä tavoin ominaisuuden toiminta todetaan oikeaksi. Mutta hyökkääjän tavoitteena on käyttää ohjelmistojen ominaisuuksia väärin, myös ohjelmiston hyväksymillä syötteillä ja tällä tavoin pyrkiä löytämään ohjelmistosta tietoturva-aukko. Tämän takia testaaminen vaatii tietoturva-asiantuntemusta.

Ohjelmistojen tietoturvaominaisuudet ovat usein niin sanottuja näkymättömiä ominaisuuksia, joiden ei ole välttämätöntä näkyä käyttäjälle juuri mitenkään. Tämä vaikuttaa myös osaltaan tietoturvaominaisuuksien kehittämiseen, koska yleensä halutaan kehittää ominaisuuksia, jotka näkyvät käyttäjille. Vaikka tietoturvaominaisuudet olisivatkin käyttäjille näkymättömiä, niin silti niiden kehitystä ei saisi laiminlyödä.

Ohjelmistojen soveltamisalueesta riippuen ei kaikkien tutkielmassa esiteltyjen vaiheiden soveltaminen tietoturvan parantamiseksi ole välttämätöntä eikä aina edes järkevää. Tutkielmassa esiteltyjen toimintatapojen tavoitteena on antaa suuntaviivoja ohjelmistojen tietoturvan parannukseen, sillä esimerkiksi pelkkien tietoturvaperiaatteiden tai tarkastuslistojen avulla voidaan saavuttaa huomattavia parannuksia ohjelmistojen tietoturvaan.

Yrityksen tietoturvauhka tulee usein yrityksen sisältä, joko työntekijöiden tiedostamattoman tai tietoisen toiminnan tuloksena. Tietoturvaa on mahdollista parantaa vaikuttamalla käyttäjien tietoturvatietoisuuteen. Tähän on mahdollista vaikuttaa esimerkiksi yrityksen tietoturvastandardit ja menettelytavat sisältävän tietoturvaoppaan avulla, sekä toimintaesimerkkien avulla. Myös ohjelmistojen tietoturvaominaisuuksien käytettävyydellä on vaikutusta ohjelmistojen tietoturvaan, sillä helppokäyttöiset ominaisuudet lisäävät tietoturvaominaisuuksien käyttöä.

Viitteet

Beattie, S., Calton, P., Cowan, C., Wagle, P., Walpole, J. (2000) Buffer overflows: attacks and defenses for the vulnerability of the decade. *DARPA Information Survivability Conference and Exposition (DISCEX '00)*, IEEE Computer Society Press, 119 – 129.

Caloyannides, M., Landwehr, G., Witten, B. (2001) Does open source improve system security? *IEEE Software* **18**(5), 57 – 61.

Gilliam, P., Sherif, J., Wolfe, T. (2003) Software Security Checklist for the Software Life Cycle. *Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaboral Enterprises (WETICE'03)*, IEEE Computer Society Press. 243 – 248.

Hoglund, G., McGraw, G. (2004) *Exploiting Software: How to Break Code*. Pearson Education Inc, United States of America (Boston). ISBN 0-201-78695-8.

Howard, M., LeBlanc, D. (2003) *Writing Secure Code 2nd ed.* Microsoft Press, Washington. ISBN 0-7356-1722-8.

Howard, M., LeBlanc, D. (2004) *Ohjelmoijan tietoturvaopas*. IT Press, Helsinki (alkuperäisen teoksen nimi: Writing Secure Code). ISBN 951-826-663-8.

Leach, J. (2003) Improving user security behaviour. *Elsevier Science Computers and Security* **22**(8), 685-692.

McGraw, G. (2002) Managing software security risks. *IEEE Computer* **35**(4), 99 – 101.

McGraw, G. (2004) Software security. *IEEE Security & Privacy Magazine* **2**(2)**,** 80 – 83.

McGraw, G., Viega, J. (2002) *Building Secure Software – How to Avoid Security Problems the Right Way*. Addison-Wesley, United States of America. ISBN 0-201-72152-X.

Neumann, P. (2004) *Principled Assuredly Trustworthy Composable Architectures*, URL: http://www.csl.sri.com/users/neumann/chats4.pdf (27.01.2005).

Nikander, P., Peltonen, T., Viljanen, L. (1996) *Internet-tietoturva*. Suomen ATK-kustannus Oy, Jyväskylä. ISBN 951-762-377-1.

Object Management Group. (2005) *UML*.WWW-sivusto, http://www.uml.org/ (13.3.2005).

Small, B. (2004) *Engineering Secure Systems Using Threat Modeling*. Software Productivity Consortium NFP Inc. URL: http://www.software.org/pub/externalpapers/sstc2004_ small2.pdf (27.01.2005).

U.S. Nuclear Regulatory Commission (1981) *Fault Tree Handbook*, NUREG-0492, Washington DC, USA. URL: http://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/ sr0492/sr0492.pdf (31.01.2005).

Viestintävirasto (2004) *CERT-FI Vuosikatsaus 2004*. URL: http://www.ficora.fi/suomi/ document/ CERT-FI_vuosikatsaus_2004.pdf (31.01.2005).

LIITE 1: Kirjautumistunnusten tarkastukseen käytettävä login.asp-tiedosto

```
<HTML>
<BODY>
<H1>
<BR><BR>
<CENTER>

<%@LANGUAGE = JScript %>
<%

function login()
{
           var fValid = false;
           var oConn = Server.CreateObject("ADODB.Connection")
           var oRS = Server.CreateObject("ADODB.Recordset")

           oConn.Open("DSN=kayttajat;UID=sa;PWD=")

           strName = Request.form("username");
           strPwd = Request.form("password");

           var strSql = "select count(*) from login where " +
           "ktunnus = '" + strName + "' " +
           " and ssana = '" + strPwd + "'";

           oRS.open(strSql, oConn);
           fValid = (oRS(0).Value > 0)
           if (fValid)
           {
%>
                      KIRJAUTUMINEN ONNISTUI
<%
           }
           else
           {
%>
                      PÄÄSY EVÄTTY
<%
           }

           oRS.close;
           delete oRS;
           oConn.Close();
           delete oConn;
           return fValid
}
           login();
%>

</CENTER>
</H1>
</BODY>
</HTML>
```