hyväksymispäivä arvosana

arvostelija

Turvallinen ohjelmistosuunnittelu

Jyrki Saarinen

Helsinki 20. huhtikuuta 2003

Tietoturvallisuus nykyaikaisessa liiketoimintaympäristössä -seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Turvallinen ohjelmistosuunnittelu

Jyrki Saarinen

Tietoturvallisuus nykyaikaisessa liiketoimintaympäristössä -seminaari
Tietojenkäsittelytieteen laitos
Helsingin yliopisto
20. huhtikuuta 2003, 19 sivua

Turvattomat eli vialliset ohjelmistotuotteet aiheuttavat nykyisin paljon rahallisia ja ajallisia menetyksiä.

Turvallisuus tarkoittaa mm. seuraavia asioita:

- Luottamuksellisuus (confidentiality) - tieto ei saa päästä asiattomien käyttäjien näkyville
- Kiistämättömyys (non-repudiation) - käyttäjä ei voi kiistää tehneensä jotakin toimintoa
- Eheys (integrity) - asiaton käyttäjä ei voi muuttaa tietoa
- Saatavuus (availability) - todennettujen käyttäjien tulee saada dataa näin halutessaan

Turvaongelmat voidaan jakaa kahteen luokkaan: määrittelyvirheisiin ja toteutusvirheisiin. Määrittelyvirheitä syntyy, kun ohjelmiston määrittelyvaiheessa turvallisuuteen liittyvät kysymykset ohitetaan tai määrittelyä ei tehdä huolellisesti. Tällaisissa tapauksissa turvallisuuteen liittyvät ominaisuudet joudutaan tekemään ohjelmistoon jälkikäteen, joka johtaa usein turvaonglemiin.

Formaalit menetelmät ovat vaatimusmäärittely- ja suunnitteluvaiheessa avuksi, sillä formaalien menetelmien avulla oletukset järjestelmästä saadaan näkyviin ja

järjestelmän toiminta voidaan spesifioida korkealla abstraktiotasolla. Spesifikaatiot ovat myös yksiselitteisiä, ja yleisesti ottaen luottamus järjestelmän oikeelliseen toimintaan kasvaa.

Suunnittelu- ja toteutustasolla järjestelmän turvallisuuteen voidaan vaikuttaa käyttämällä tiettyjä välineitä. Esimerkiksi Java-ohjelmointikieli on turvallisempi vaihtehto verrattuna C-ohjelmointikieleen, sillä Java-ohjelmointikielen muistinhallinta ei mahdollista puskurin ylivuotoon perustuvia hyökkäyksiä.

Luokittelemalla jo löydetyt virheet tietyllä tavalla voidaan alentaa ohjelmiston ylläpitokustannuksia. Tällaisen luokittelun perusteella voidaan ennustaa ja ehkäistä turvallisuusongelmia nykyisissä ja tulevissa ohjelmistoissa. Tällaista luokittelua voidaan hyödyntää myös testausvaiheessa.

Aiheluokat(Computing Reviews 1998): D.2, D.4.6, K.6.3

Avainsanat: tietoturva, ohjelmistotuotanto, testaus

Sisältö

1 Johdanto **1**

1.1 Turvaongelmien sijoittuminen 2

2 Määrittely **2**

3 Suunnittelu **4**

4 Totetus **6**

4.1 Syötteiden validointi 7

4.2 Puskurin ylivuodon välttäminen 7

4.3 Ohjelman rakenne 9

5 Testaus **11**

5.1 Korat 12

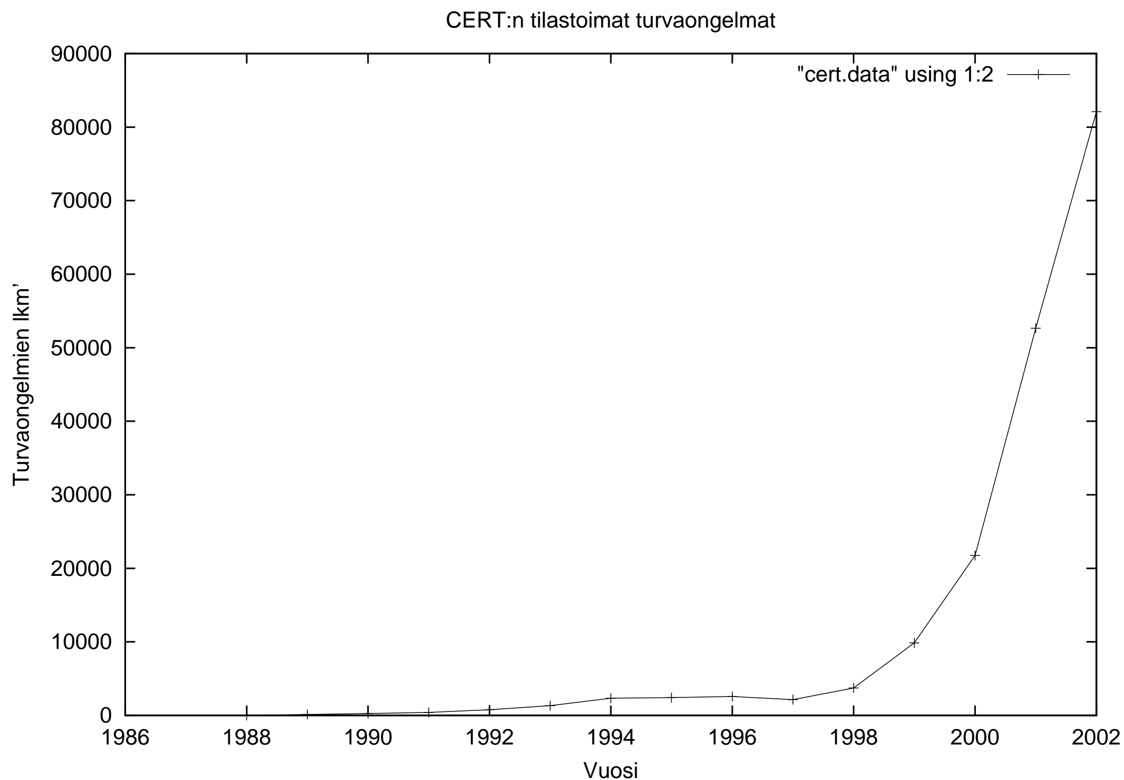
5.2 JTest 15

6 Yhteenveto **18**

Lähteet **18**

1 Johdanto

Turvattomat eli virheelliset ohjelmistotuotteet aiheuttavat nykyisin paljon rahallisia ja ajallisia menetyksiä. Kuvassa 1 on CERT/CC:n [CERT] tilastoimien turvaongelmien (incident) määräät vuosina 1989-2002. Trendi on selvästi nouseva. Tämä johtunee siitä, että nykyään ohjelmistoja käytetään laajemmalti, kuin esimeriksi vuonna 1989. Lisäksi yhä useampi tietokone jossa ohjelmistoja suoritetaan on kiinni julkisessa verkossa, Internetissä. Ohjelmistot eivät siis ole huonolaatuisempia kuin aikaisemmin. Turvaongelmia aiheuttavien ohjelmistovikojen havaitsemiseen on sen sijaan alettu viime vuosina alettu kiinnittää enemmän huomiota. Tämä on johtunut sekä kasvaneesta tietoisuudesta että vihamielisistä turvaongelmien etsijistä ja hyödyntäjistä (krakkerit).



Kuva 1: CERT/CC:n tilastomat turvaongelmat

Tässä esitelmässä käydään ohjelmistotuotannon päävaiheet läpi siten, että jokaisen vaiheen kohdalla esitetään menetelmiä ja keinoja joita käyttämällä ohjelmistoprosesin tuloksena olisi turvallisempi ohjelmisto. Määrittelyvaiheessa voidaan hyödyntää formaaleja menetelmiä. Ohjelmiston vaatimukset ilmaistaan jollain riittävän muodollisella tavalla. Suunnittelu- ja toteusvaiheessa voidaan käytettävien välineiden valinnalla on suuri merkitys. Testausvaiheessa voidaan hyödyntää testaustyökaluja. Lisäksi, jos ohjelmiston vaatimukset oli määritelty formaalisti, voidaan usein testitapaukset (test suite) johtaa näistä vaatimuksista automaattisesti.

1.1 Turvaongelmien sijoittuminen

Tunnetusti ohjelmistossa tai muussa ohjelmistoprosessin osavaiheen tuotteessa esiintyvän vian korjaamisen kalleus on suoraan verrannollinen vian syntymis- ja havaitsemisajankohtien erotukseen. Jos siis ohjelmistossa havaittu turvaongelma johtuu virhestä vaatimusmäärittelyssä (protokollassa ei ole todentamista (authentication), vaikka sen piti olla mukana) on sen korjaaminen paljon kalliimpaa kuin toteutusvaiheessa syntyneen ongelman (puskurin ylivuoto) korjaaminen. Turvaominaisuuksien lisääminen ohjelmistoon jälkikäteen 'ad-hoc' ei myöskään useimmiten johda hyvään lopputulokseen [DeSt00]. Onneksi suurin osa turvaongelmista johtuu puskurin ylivuodoista [Sek99]. Puskurin ylivuoto-ongelmat voidaan usein korjata nopeasti ja niiden syntyminenkin on helposti estettävissä.

2 Määrittely

Ohjelmiston vaatimusmäärittelyvaiheessa pyritään dokumentoimaan ohjelmiston haluttu toiminta mahdollisimman tarkasti. Esimerkiksi osa verkossa toimi-

van tiedostopalvelimen toimintaa voitaisiin määritellä näin: 'jokaisen tiedostopalvelimen käyttäjän tulee todentaa itsensä ennen palvelun saamista'. Tämä vaatimus on varsin selkeästi ilmaistu - usein ohjelmistovaatimukset ovat ylimalkaisia ja jopa keskenään ristiriitaisia. Erilaisten formalismien käytöllä pyritään vaatimusten täsmällisempään ilmaisuun. Sen lisäksi, että näitä formaalisti ilmaistuja vaatimuksia voidaan prosessoida ja hyödyntää ohjelmistoprosessin muissa vaiheissa, niiden tuottaminen vaatii tarkkuutta, ja ehkä näin ilmaistuissa vaatimuksissa on tuotu ne implisiittiset oletukset esiin mitä sanallisesti ilmaistuissa vaatimuksissa oli [Hir90].

Formalismien ajatuksena on rakentaa järjestelmän toiminnasta kuvaus jollakin muodollisella tavalla eli muodostaa järjestelmän *spesifikaatio*. Toteutettu järjestelmä *verifioidaan* spesifikaatiotansa vasten matemaattisin menetelmin. Tämän prosessin tuloksena on täsmällinen vastaus siihen, toteuttaako järjestelmä sille annetut vaatimukset vai ei [1]. Etu testaukseen verifiointimenetelmänä on huomattava: verifiointi on suoritettavissa tietokoneella (automaatinen testaus), kun taas testauksessa tarvitataan usein ihmistyötä testitapausten muodostamisessa. Testauksen lopussa ollaan testauksen laajuudesta riippuen enemmän tai vähemmän luottavaisia, että järjestelmä toimii spesifikaationsa mukaisesti, kun taas formaaleja menetelmiä käytettäessä vastaus on täsmällinen.

Useat formalismit, kuten Petri-verkot ja aikalogiikka, ovat omimmalla alueellaan, kun niitä sovelletaan hajautettujen ja rinnakkaisten järjestelmien (tietoliikenneprotokollat jne.) parissa. Edellä mainitun ohjelmistovaatimuksen 'jokaisen tiedostopalvelimen käyttäjän tulee todentaa itsensä ennen palvelun saamista' ilmaisu tällaisilla formalismeilla voi olla hankalaa ja epäkäytännöllistä. Vaatimus onkin ilmaistavissa yksinkertaisesti predikaattilogiikan keinoin: $isRequest(p) \rightarrow$

[1] Usein saadaan myös vastaus kysymykseen 'miksi ei?' jos järjestelmä ei toteuta spesifikaatioaan

$authenticated(p.user)$ (p on palvelimelle saapunut sanomat ja $p.user$ viittaa sanoman lähettäjän identiteettiin).

3 Suunnittelu

Suunntteluvaiheessa määritellään ohjelmiston modulien rajapinnat. Näiden rajapintojen toiminnan määrittely on tehty jo määrittelyvaiheessa äidinkielen lausein tai predikaattilogikaan keinoin. Modulien rajapintoja suunnitellessa ja dokumentoitaessa, voidaan näiden vaatimusten huomiointi varmistaa kääntämällä vaatimukset jollekin sopivalle soesifiointikielelle. Java Modelling Language (JML) [JML] on yksi tapa määritellä Java-kielisten ohjelmien käyttäytymistä muodollisella tavalla. Edellisen kappaleen $isRequest(p) \rightarrow authenticated(p.user)$ on ilmaistavissa paketteja käsittelevän modulin rajapinnassa JavaDoc-dokumentaationa seuraavasti:

```
public interface Processor
{
  /*@
    @ public normal_behaviour
    @   requires packet.isRequest()
    @   ensures authenticated(packet)
   */
  public void processPacket(Packet packet);
}
```

Edellinen JML-spesifikaatio siis sanoo, että rajapintakutsun $processPacket$ toiminta on normaalitilanteessa ($normal_behaviour$) seuraava: jos predikatti (eli metodi) $packet.isRequest()$ on voimassa (eli palauttaa $true$), niin $authenticated(packet)$ on oltava myös tosi. Toisin sanoen $normal_behaviour$ määrittelee, että implikaation $requires \rightarrow ensures$ on oltava tosi ellei metodista $processPacket$ heitetä $java.lang.RuntimeEx$ tai $java.lang.Error$:n alipoikkeusta.

Loogiset lauseet $requires$ ja $ensures$ koostuvat siis predikaateista eli Java-meteodeista ja niitä yhdistävistä loogisista konnektiiveista (ja, tai, jos-niin, ei, joss). Näitä lauseita kutsutaan myös *esiehdoksi* (precondition) ja *jälkiehdoksi* (post condition). Esiehto kertoo jotakin olion oikeellisesta tilasta, ja jos esiehto on voimassa ennen metodin suoritusta, on jälkiehdonkin oltava voimassa metodin suorituksen jälkeen. Lisäksi voidaan määrittää kolmas looginen lause, *luokkainvariantti*, kertomaan luokan ilmentymien oikeellisesta tilasta.

JML:n kaltainen mentelmä on Design by Contract [JTest]. Seuraavassa esimerkissä olisi määriteltynä Design by Contract -menetelmällä luokka $Ihminen$ ja sille luokkainvariantti 'Jokaisen ihmisen massa on positiivinen'. Lisäksi konstruktorissa on määritelty, että ihmisellä täytyy olla vähintään etu- ja sukunimet.

```
/**
 @ inv massa > 0
 */
public class Ihminen
{
  /**
   * [0] = Sukunimi, [1] = etunimi, [2]..[n] toisen etunimen osat
   */
  private String[] nimenKomponentit;
  private double massa;

  /**
   @ pre nimenKomponentit.length >= 2
   */
  public Ihminen(double massa, String[] nimenKomponentit)
  {
    this.massa = massa;
    this.nimenKomponentit = nimenKomponentit;
  }
}
```

Predikaatit ovat siis tavallisia Java-lauseita. Lisäksi voidaan käyttää universaali- (∀) ja ekstitentiaalikvanttoreita (∃). Seuraavassa esimerkissä määritellään, että $getEmployees()$:n palauttaman kokoelman (java.util.Collection) jokaisella työntekijällä on $getRooms()$:n palauttamssa kokoelmassa huone:

```
/*
 @inv forall Employee e in getEmployess() | getRooms().contains(e.getOffice())
 */
```

Design by Contract -menetelmässä voidaan myös ottaa kantaa metodin samanaikaisuuden hallintaan @$concurreny$-lauseella. Erilaisia rinnakkaisuusmääreitä ovat 'concurrent' (metodi voidaan suorittaa rinnakkain), 'guarded' ja 'sequential' (useampi säie ei saa suorittaa metodia samanaikaisesti). Metodin poikkeuksellinen käyttäytyminen kuvataan @$exception$-lauseella.

4 Totetus

Toteutusvaiheessa voidaan käytettävillä työkaluilla - kuten käytettävällä ohjelmointikielellä - vaikuttaa suuresti siihen, syntyykö tässä ohjelmistoprosessin vaiheessa turvaongelmia ja että pystytäänkö määrittelyvaiheessa syntyneet vaatimukset toteuttamaan. Ohjelmiston toteutuskieleksi voitaisiin valita Java C-kielen sijaan. Tällöin vältytään puskurin ylivuodosta aiheutuvilta turvaongelmilta sillä Java-ohjelmointikieli ei mahdollista puskurin ylivuotoa. Aina tällainen ei ole tietenkään mahdollista, voi olla että on pakko käyttää esimerkiksi C-kieltä ainakin osittain. Tällöin voidaan käyttää lähde- ja objektikoodin analysointiohjelmia, jotka etsivät puskurin ylivuotoon mahdollisesti johtavia rakenteita [Ba02].

Turvallisen ohjelmistosuunnittelun oleellisimpia toteutuskohtaisia asioita ovat syötteiden validointi, puskurin ylivuodon välttäminen sekä vähmmäisoikeuksien periaate [Whe03]. Jos ollaan toteuttamssa sen tyyppistä ohjelmistoa, jonka tekemi-

sestä on historiatietoa ja kokemusta, voidaan vikojen sijainti mahdollisesti ennustaa [JiZ02].

4.1 Syötteiden validointi

Hyvin suuri turvaongelmista aiheutuu siitä, ettei ohjelmistossa tarkisteta sille tulevia syötteitä riittävän hyvin (tai ollenkaan). Jokainen ohjelmiston ulkopuolelta tuleva syöte tulisi siis validoida. Validointi tulisi tapahtua siten, että muodostetaan säännöt, jotka laillisen syötteen täytyy täyttää. Kaikki muut syötteet ovat epäkelpoja. Syötteiden validointi siten, että muodostetaan säännöt epäkelvoille syötteille, ja hyväksytään loput, on vaarallista [Whe03].

Merkkijonosyötteet voidaan usein validoida muodostamalla säännöllinen lauseke, jolla voidaan tunnistaa kelvolliset merkkijonot. Jos käytössä on Unicode-merkkijonot ja erityisesti niiden UTF-8 -esitysmuoto, täytyy UTF-9 -dekoodaajissa varautua rikkinäisiin UTF-8-merkkijonoihin. Tiedostonimien ja yleisemmin URI:en käsittelyyn liittyy '../' eli viittaus ylempään hakemistoon voi olla vaarallinen. Tällaista konstruktiota hyödyöntämällä hyökkääjä voi saada luettua esimerkiksi W^3-palvelimen tiedostojärjestelmästä haluamiaan tiedostoja (/etc/passwd jne.).

4.2 Puskurin ylivuodon välttäminen

Suurin osa turvaongelmista näyttäisi aiheutuvan puskurin ylivuodosta. Tällä tarkoitetaan tilannetta, jossa puskuriin luetaan enemmän kuin puskuriin mahtuu. Jos puskuri on pinosta varattu muistialue, voi tätä vihamielinen hyökkääjä suorittaa omaa koodian puskurin ylivuodosta kärsivän ohjelmiston oikeuksilla kohdejärjestelmässä. Tämä tapahtuu siten, että puskurin ylivuodon aiheuttavaan syötteeseen laitetetaan suoritettavaksi haluttavan konekoodin osoite sellaiseen kohtaan, että se ylikirjoittaa pinossa aliohjelman paluuosoitteen (return address). Alioh-

jelmassta poistuttaessa ei siis palatakaan sinne, josta aliohjelmaa kutsuttiin, vaan siirrytään suorittamaan hyökkääjän omaa ohjelmakoodia [On96]. On myöskin käyttöjärjestelmiä, joissa koodin suoritus pinossa on estetty tai on ylläpitäjän estettävissä.

Seuraavassa C-kielisessä esimerkissä luetaan puskuriin $stdin$-tiedostovirrasta tavuja standardikirjaston $gets$-kutsulla: $gets$ lukee $stdin$:stä tavuja parametrina annettuun puskurin alkuosoitteeseen rivinvaihtoon tai tiedoston loppumerkkiin EOF asti:

```
#include <stdio.h>

#define BUFFER_MAX 64
int main(int argc, char **argv)
{
  unsigned char buffer[BUFFER_MAX];
  gets(buffer);
  do_something_with_the_buffer(buffer);
  return 0;
}
```

Koska $gets()$:llä ei voi olla $buffer$-osoittimien osoittaman puskurin koosta minkäänlaista tietoa, sillä puskurin koko ei välitetä sille, voi puskurin ylivuoto tapahtua jos vihamielinen käyttäjä syöttää tämän ohjelman $stdin$-tiedostovirtaan sopivan syötteen. Lisäksi hyökkääjän täytyy tietää hyökkäyksen kohteen arkkitehtuuri, jotta puskurin ylikirjoittavassa syötteessä oleva haluttu konekielinen koodi tulisi suoritettua.

Tämän puskurin ylivuodon aiheuttavan ongelman korjauksena on käyttää $fgets$-kirjastokutsua $gets$:n sijaan. $fgets$:lle annetaan myöskin puskurin pituus parametrina, ja näin ollen se pystyy lopettamaan puskuriin kirjoittamisen tilan loppuessa. Korjattu versiot:

```
#include <stdio.h>

#define BUFFER_MAX 64
int main(int argc, char **argv)
{
  unsigned char buffer[BUFFER_MAX];
```

```
  fgets(buffer, 0, BUFFER_MAX, stdin);
  do_something_with_the_buffer(buffer);
  return 0;
}
```

Vastaavia turvattomia kirjastofunktioita C-standardikirjastosta löytyy muitakin, esimerkiksi $sprintf$. Tässä puskurin ylivuotoesimerkissä kyse oli jo lähtökohtaisesti turvattoman kirjastofunktion käytöstä. Puskuri voi vuotaa yli monella muullakin tavalla: on protokolla, jossa on määritelty tietopaketeille maksimikoko MAX. Jos protokollan toteuttaja uskoo kaikkien toimivan tämä määrittelyn mukaisesti, voi puskurin ylivuodon hyväksikäyttö tulla mahdolliseksi. Toteutuksessa voidaan toki varata kiinteä MAX kokoinen puskuri pinosta, ja lukea siihen tietovirrasta (esim. TCP-pistokkeesta) koko ajan tarkisten ettei kirjoiteta puskurista yli. Ei siis voi luottaa, että kukaan ei lähettäisi enempää kuin MAX tavua suurempia paketteja. Tämä kyseinen esimerkki liittyy oikeastaan enemmän edellän mainittuun syötteiden tarkistamiseen.

C++-ohjelmointikieltä käytettäessä voidaan puskurin ylivuotoa välttää käyttämällä standardikirjaston merkkijonototeutusta ($std :: string$) ja STL:n tarjoamia tietorakenteita ja säilöitä [Whe03].

Toinen vaihtoehto on käyttää ohjelmointikieltä, joka ei mahdollista puskurin ylivuotoa. Tällaisia ohjelmointikieliä ovat esimerkiksi Java, Python, Perl ja Ada95. Java:ssa puskurin ylikirjoitus havaitaan $java.lang.ArrayIndexOutOfBoundsException$-poikkeuksen lentämisenä.

4.3 Ohjelman rakenne

Seuraavassa luetellaan muutmia yleisperiaatteita joita tulisi soveltaa turvallista ohjelmistoa toteutettaessa:

Vähimmäisoikeuksien periaate. Ohjelman toteutuksessa tulee käyttää niin pieniä käyttöoikeuksia kuin on mahdollista. Jos ohjelman tarvitsee nostaa käyttöoikeuksiaan tehdäkseen jonkin asian, käyttöoikeuksien ei tule olla nostettuna kuin tarpeellisen ajan.

Rajapintojen turvallisuus. Rajapintojen tulisi olla minimaalisia siten, että ne tarjoavat vain tarvittavat funktiot eivätkä mitään muuta. Rajapintojen tulisi myös olla ohittamattomia (non-bypassable).

Datan ja kontrollin erottaminen. Ohjelmiston lukeman tiedostoformaatin tulisi olla täysin passivinen. Tämä tarkoittaa sitä, että tiedosto sisältää ainoastaan tietoa, ei sulautettuja ohjelmia (makroja tai skriptejä).

Samanaikaisuudesta aiheutuvien ongelmien välttäminen. Esimeriksi jaettujen muuttujien käytössä tarvitaan keskenäistä poissulkemista. Jos poissulkemista ei ole hoidettu, voi vihamielinen prosessi aiheuttaa turvaongelman syöttämällä toiselle prosessille dataa. Poissulkemisen väärintoteuttamisesta taas voi seurata palvelunestotilanne (Denial of Service) prosessin lukkiutuessa (deadlock).

Luotettavat kanavat. Julkisissa tietokoneverkoissa luotettava kanava on sellainen, joka säilyttää tiedon eheyden ja luottamuksellisuuden (integrity, confidentiality) ja että tiedon lähettäjä on todennettu (authentication). Epäluotettavasta kanavasta tulevaan tietoon ei ole syytä luottaa. Luotettava kanava saadaan aikaan kryptografian keinoin (julkisen avaimen menetelmä mahdollistaa käyttäjän todentamisen, salaisen avaimen menetelmillä saadaan aikaan tiedon eheys ja luottamuksellisuus). IP-osoitteita ja/tai portteja ei voida käyttää pääsynvalvontaan, sillä niiden väärentäminen on helppoa eivätkä IP-osoitteet voivat olla dynaamisia nykyään (DHCP, liikkuvat työasemat jne.).

5 Testaus

Yksi tapa suorittaa verifiointia - eli sitä aktivitettia jonka tuloksena saadaan vastaus siihen, toimiiko ohjelmisto spesifikaationsa mukaisesti - on testaus. Testaus on kallista ja aikaaviepää, ja on oma aliprosessina ohjelmistoprosessissa. Ensin tehdään testaussuunnitelma, jossa tärkeimpänä päätetään mitä testataan (moduli, luokka, kirjasto), miten testataan (black-box, white-box) ja missä vaiheessa testaus katsotaan suoritetuksi (kaikki testitapaukset suoritettu, 90% haaraumakattavuus saavutettu). Tämän jälkeen luodaan testitapaukset, suoritetaan ne ja lopuksi kirjataan havaitut viat ja puutteet.

Toinen tapa suorittaa verifiointia oli formaalien menetelmien käyttö, jossa järjestelmän käyttäytyminen oli määritelty täsmällisesti. Menetelmään liittyvällä algoritmilla voidaan täsmällisesti tarkistaa, toteuttaako järjestelmä ne ominaisuudet, joita siltä vaadittiin. Testausta ei siis tarvittu, vaan järjestelmän oikeellisuus todistettiin (todistus tapahtuu usein algoritmisesti koneella). Valitettavasti näin voidaan ohjelman oikeellisuudesta varmistua vain osajoukossa ohjelmia.

Testitapausten suoritus, tulosten analysointi ja regressiotestaus ovat kohtalaisen helposti automatisoitavissa skriptikieliä käyttämällä. Esimerkkjä testauksessa hyödyllisistä työkaluista ovat esimerkiksi $Perl$, $Python$, sh, awk ja sed. Java-ohjelmien yksikkötestauksen automatisoinnin mahdollistaa JTest [JTest]. JTest on ohjelmistokehys joka tarjoaa hyvin yksinkertaisen tavan tehdä yksikkötestejä ja suorittaa niitä.

Testitapausten luominen on testausprosessin raskain osuus. Onneksi se onkin automatisoitavissa, jos ohjelman toiminta on määritelty tarpeeksi täsmällisesti, esimerkiksi käyttäen edellä mainittuja JML- ja Design by Contract -menetelmiä. Tällöin voidaan ohjelman käyttäytymisen spesifikaatiosta tuottaa testitapaukset, suorittaa nämä testitapaukset ja raportoida testaajalle täyttikö ohjelma spesifikaa-

tionsa, ja jos ei, niin miksi [2]. Luonnollisesti ohjelman toivottavan käyttäytymisen määrittävien spesifikaatioiden tulisi olla oikein tehty: niiden ristiriidattomuudesta voidaan varmistua logiikan keinoin ja semanttisesta järkevyydestä voidaan olla hieman varmempia katselmointien ym. staattisen [Tai02] testauksen menetelmien avulla.

5.1 Korat

Korat [BKM02] on ohjelmistokehys, jolla testataan Java-ohjelmia black-box -tekniikoin. Se osaa tuottaa metodin (aliohjelman) testitapaukset automaattisesti, kun metodin käyttäytyminen on määritelty JML-spesififikaation avulla.

Testitapaukset Korat tuottaa seuraavsti: metodin esiehdosta tuotetaan *Java-predikaatti*, eli metodi joka palauttaa totuusarvon. Tämän jälkeen Korat tuottaa kaikki ne metodin ei-isomorfiset syötteet joille Java-predikaatti palauttaa arvon *tosi*.

Seuraavassa esimerkissä käsitellään binääripuuta. Java-predikaattina on metodi *repOk*, joka palauttaa toden jos kyseessä on puu (ei syklejä). Kyseinen predikaatti on myöskin $BinaryTree$-luokan invariantti.

```
import java util.*;
class BinaryTree
{
  private Node root; // juuri
  private int size;  // puun solmujen määrä
  static class Node  // solmu-luokka
  {
    private Node left, right;
  }
  // luokkainvariantti
  public boolean repOk()
  {
```

[2] Tällaista automaattista testausta ei kai pitäisi enää laskea testauksen piiriin, vaan spesifioinnin ja verifioinnin piiiriin

```
    // tyhjän puun koko on nolla
    if (this.root == null)
      return this.size == 0;
    Set visited = new HashSet();
    visited.add(this.root);
    LinkedList workList = new LinkedList();
    workList.add(this.root);
    while (!workList.isEmpty())
    {
      Node current = (Node)workList.removeFirst();
      if (current.left != null)
      {
        // tarkastetaan onko sykli
        if (!visited.add(current.left))
          return false;
        workList.add(current.left);
      }
      if (current.right != null)
      {
        // tarkastetaan onko sykli
        if (!visited.add(current.right));
          return false;
        workList.add(current.right);
      }
    }
    // koon tulisi olla konsistentti
    return visited.size() == this.size ? true : false;
  }
}
```

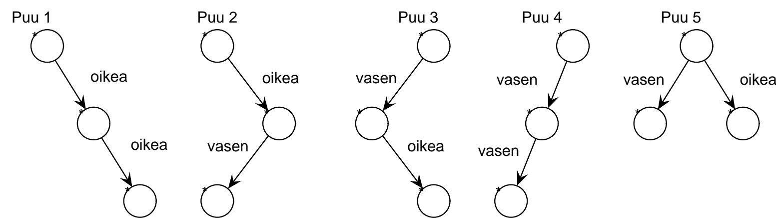
Korat:n hakualgoritmi tuottaa n:n kokoiset keskenään ei-isomorfiset binääripuut *äärellistämiskuvaksella* (finitization description). Äärellistämiskuvauksella määritellään kaikki maksimissaan tietyn kokoiset binääripuut:

```
public static Finitzation finBinaryTree(int numNodes)
{
1:  Finitization f = new Finitization(BinaryTree.class);
2:  ObjSet nodes = f.createObjects("Node", numNodes);
3:  nodes.add(null);
4:  f.set("root", nodes);       // root \in (null + Node)
```

```
5:  f.set("size", numNodes);    // size
6:  f.set("Node.left", nodes);  // Node.left \in (null + Node)
7:  f.set("Node.right", nodes); // Node.right \in (null + Node)
8:  return f;
}
```

Rivillä 1 luodaan binääripuuta kuvaava olio. Rivillä 2 luodaan kaikkien solmujen joukko. Rivillä 3 kerrotaan, että olematon solmu ($null$-osoitin) on mahdollinen alipuun juuri. Riveillä 4-7 kerrotaan mitä mahdollisia arvoja luokan $BinaryTree$ kentät voivat saada.

Kuvassa 2 on Korat:n tuottamat kaikki kolmen solmun kokoiset keskenään ei-isomorfiset binääripuut. Olkoon tämän joukon nimi B. Kaksi binääripuuta ovat isomorfiset, joss haarautumisrakenne on sama. Solmujen varsinainen sisältö ei siis vaikuta isomorfismin olemassaoloon.



Kuva 2: Korat:n algoritmin tuottamat ei-isomorfiset binääripuut (finBinaryTree(3))

Seuravaaksi määritellään binääripuun alkion poistometodin toiminta metodin formaalilla spesifikaatiolla.

```
//@ public invariant repOk(); // luokan invariantti

/*@ public normal_behaviour   // poiston spesifikaatio
  @    requires has(node);    // esiehto
  @    ensures !has(node);    // jälkiehto
  @*/
public void remove(Node node)
```

```
{
  // poistometodin koodi
}
```

Metodin spesifikaatio olennaisesti sanoo, että normaalisti $remove$-metodia ennen poistettava solmu oli puussa ja poiston jälkeen sitä ei enää ole puussa. Implikaation $has(node) \rightarrow !has(node)$ on aina voimassa, jos $remove$ on toteutettu oikein.

Metodin testaus tapahtuu seuraavasti: Korat tuottaa testitapausjoukon T: $(t = (b, n) \in T \iff b \in B \wedge n \in N)$, missä N on kaikkien B:ssä olevien puiden solmujen joukko mukaanlukien tyhjä solmu ($null$-osoitin). Tämän jälkeen poistometodi suoritetaan jokaisella testitapauksella $t \in T$. Jos jokin testitapaus ei toteuta metodin spesifikaatiota, Korat raportoi tämän testitapauksen käyttäjälle vastaesimerkkinä.

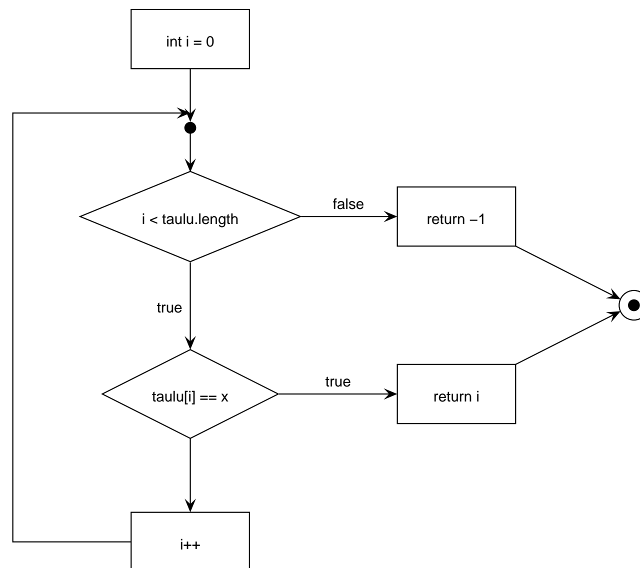
5.2 JTest

JTest on Parasoft-nimisen yrityksen tuottama kaupallinen testaustyökalu [JTest]. JTest tuottaa testitapaukset automaattisesti, ajaa testitapaukset ja raportoi testien tuloksista käyttäjälle. Luokan rakenteellista testausta suoritetaan white-box -menetelmin. Toiminnallista testausta suoritetaan Design by Contract -menetelmällä, jossa metodin käyttäytyminen määritellään Korat:n tapaan formaalilla spesifikaatiolla. JTest tukee myös regressiotestausta raportoimalla testitulosten erot edelliseen testauskierrokseen. JTest osaa myös valvoa koodausstandardeja sekä toimii erilaisten ohjelmistomittojen raportoijana.

Luokan eli ohjelmakoodin rakenteen JTest testaa white-box -menetelmin. Javakääntäjällä luokan lähdekoodista tuotetaan tavukoodi. Tästä tavukoodista JTest voisi muodostaa (JTest:n käyttämää algoritmia ei ole kuvattu ja se on patentoitu) metodien kontrollivuoverkot, joissa verkon solmut kuvaavat ohjelman lauseita ja

kaaret kontrollin siirtymistä lauseiden välillä [Tai02]. Seuraavan esimerkkiohjelman kontrollivuoverkko on kuvassa 3.

```
/**
 * Linearihaku.
 * @param x
 *   Haettava alkio
 * @param taulu
automaattinen.tex lines 169-196/3 *   Taulukko, josta haetaan. EI saa
olla null.
 * @return
 *   Haetun alkion indeksi taulukossa tai -1 jos ei löytynyt.
 */
public static int lineaarihaku(int x, int[] taulu)
{
  for (int i = 0; i < taulu.length; i++)
    if (taulu[i] == x)
      return i;
  return -1;
}
```



Kuva 3: Lineaarihaun kontrollivuoverkko

Tässä tapauksessa JTest voisi tuottaa seuraavat testitapaukset:

1. x, <null>
2. x, <0:n mittainen taulu>
3. x, <taulu jossa on x>
4. x, <taulu jossa ei ole x>

Testitapaukset tuotettuaan JTest ajaa ne. Jos testattava metodi 'kaatuu', raportoidaan kaatumisesta käyttäjälle. Java-ohjelman yhteydessä 'kaatuminen' tarkoittaa sitä, että metodi heittää (throws) ns. uncaught-poikkeuksen. Tällaisten uncaught-poikkeusten sieppaamista (catch) Java-kääntäjä ei pakota ohjelmoijaa tekemään, joten metodista lentävä poikkeus jää yleensä JVM:n käsiteltäväksi. Tässä mielessä voidaan puhua 'kaatumisesta', sillä poikkeuskäsittely ei ole hallittua [3].

White-box -testauksen kattavuutta arvioidaan erilaisin mitoin, kuten lause- ja haaraumakattavuus [Tai02] (statement coverage, branch coverage). Edellämainituilla testitapauksilla saavutetaan kontrollivuoverkossa 3 selvästi täysi (100%) lause- ja haaraumakattavuus. JTest on tosin ensimmäisen testitapauksen kohdalla löytävinään virheen ohjelmakoodista (ensimmäiseltä koodiriviltä lentää *NullPointerException*, mutta metodin dokumentaatio sanoo ettei tätä metodia tule kutsua *null*-osoittimella). Tällaiset ongelmat saattavat hämärtää testitulosten tulkitsemista.

Luokan toimintaa (functional testing) JTest testaa black-box-menetelmällä. Testitapausten tuottamista ohjaa Design by Contract-niminen spesifiointitekniikka, jossa metodin toiminta spesifioidaan formaalisti kuten Korat:n tapauksessa.

[3] Jos uncaught-poikkeuksen heitti viimeinen elossa oleva ei-daemon-säie, koko JVM-prosessi poistuu ja koko ohjelma (prosessi) 'kaatuu'.

6 Yhteenveto

Turvallisen ohjelmistosuunnittelun keinot ja menetelmät ulottuvat jokaiselle ohjelmistotuotantoprosessin osa-alueelle. Mitä aikaisemmin ohjelmistprosessissa turvaongelmat syntyvät, sen vaikeampi niitä on korjata jälkikäteen. Suurin osa turvaongelmista on onneksi nykyisin puskurin ylivuodosta johtuvia toteutusteknisiä ongelmia, joiden korjaaminen on halvempaa kuin tietoturvaominaisuuksien jälkikäteen lisääminen ohjelmistoon.

Formaalit menetelmät mahdollistavat ohjelmiston ominaisuuksien täsmällisen määrittelyn vaatimusmäärittelyvaiheessa. Toteutuksen oikeellisuus voidaan joskus verifioida algoritmisesti todistaen toteutus spesifikaationsa mukaiseksi.

C- ja C++-ohjelmointikielet mahdollistavat puskurin ylivuodosta aiheuvan turvaongelman syntymisen. Näiden ohjelmointikielten sijaan voidaan käyttää turvallisempia ohjelmointikieliä. Niiltä osin joille C:n tai C++:n käyttö on tarpeellista, voidaan puskuriylivuodot eliminioita erilaisin ohjelmointitekniikoin.

Testaukseen on saatavissa monia testaustyötä helpottavia työkaluja. Erityisesti aikaisemmassa vaiheessa luotua ohjelman käyttäytymisen formaalia spesifikaatiota voidaan hyödyntää automaattisessa testauksessa.

Lähteet

Ba02 M. Bauer. *Paranoid peguin: Q&A with Chris Wysopal (weld pond)*. Linux Journal, September 2002, Volume 2002, Issue 101.

BKM02 C. Boyapati, S. Khurshid, D. Marinov, *Korat: automated testing based on Java predicates*, ACM SIGSOFT Software Engineering Notes, 2002.

CERT Computer Emergency Response Team / Coordination Center, http://www.cert.org [20.4.2003].

DeSt00 P.T., Devanbu, S., Stubblebine. *Software engineering for security: a roadmap*. Proceedings of the conference on The future of Software engineering, May 2000.

Hir90 G.R., Hird. *Formal method in software engineering*. Proceedings of Digital Avionisc Systems Conference, Oct 1990.

JiZ02 K., Jiwnani, M., Zelkowitz. *Maintaining software with a security perspective*. Proceedings of International Conference on Software Maintenance, 2002.

JML Java Modelling Language, http://www.cs.iastate.edu/~leavens/JML.html [20.4.2003].

JUnit JUnit, http://www.junit.org [20.4.2003].

JTest JTest, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest&itemId=11 [20.4.2003].

On96 Aleph One, *Smashing The Stack For Fun And Profit*, http://www.shmoo.com/phrack/Phrack49/p49-14 [20.4.2003].

Sek99 R. Sekar, *Classification of CERT/CC Advisories 1993-1998*. http://seclab.cs.sunysb.edu/sekar/papers/cert.htm [20.4.2003].

Tai02 Juha Taina, *Ohjelmistojen testaus -kurssin luentomateriaali*, TKTL, Helsingin yliopisto, 2002.

Whe03 David A. Wheeler, *Secure Programming for Linux and Unix HOWTO*. http://www.dwheeler.com/secure-programs/ [20.4.2003].