Juhani Paavilainen (toim.)

Tietoturvallinen ohjelmointi

DEPARTMENT OF COMPUTER SCIENCES

UNIVERSITY OF TAMPERE

B-2004-5

TAMPERE 2004

Sisältö

Alkusanat

Tämä julkaisu koostuu Tampereen teknillisen yliopiston ”tietojenkäsittelyn turvallisuus” – seminaarissa ja Tampereen yliopistossa järjestetyn ”Tietoturvallinen ohjelmointi” - seminaarien töistä. Vaikka osallistujia seminaareissa oli vähän, oli osallistujien työmotivaatio hyvä ja osa seminaaritöistä on erittäin hyviä. Kiitokset tästä seminaarin osallistujille.

Lähtökohtana seminaarissa oli Mark G. Graff ja Kenneth R. van Wyk kirja ”Secure Coding: Principles & Practices”. Seminaarilaiset tekivät jokainen tiivistelmät tietyistä kirjan luvuista. Koska kirjasta puutui kriittisten järjestelmien ohjelmistotuotannon yleisperiaatteet, otettiin mukaan Ian Somervillen ”Software Engineering” kirjasta tarvittavat osat, jotka toimivat johdantona ja teoreettisena viitekehyksenä aihealueeseen.

Julkaisun tarkoituksena on toimia alustuksena tuleville tietoturvallisuuden ohjelmointikursseille ja seminaareille. Julkaisu on siinä mielessä harvinainen, että tietoturvalliseen ohjelmointiin ei ole olemassa suomenkielistä kirjallisuutta juuri ollenkaan vaikka aihetta sivutaankin monessa kirjassa. Toivomme että tämä julkaisu osaltaan täyttää olemassa olevaa tyhjiötä.

Juhani Paavilainen

1. Tietoturvallinen ohjelmointi ja ohjelmistonkehitys

Juhani Paavilainen

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

1.1. Yleistä

Teksti perustuu pääosin kirjaan: Ian Sommerville, Software Engineering, 6th ed., Part 4 Critical Systems, Chapters 16..18 (p 353…417). ja Part 5, Verification Validation, Chapters 19..21, p. 417…486.

Tekstissä ei ole otettu kriittistä kantaa Somervillen esittämiin argumentteihin. Toisaalta siihen on hyvin vähän tarvetta, koska alkuperäinen teksti perustuu hyvin dokumentoituihin tieteellisiin tutkimuksiin. Näihin lähteisiin ei viitata enää tässä tekstissä vaan ne ovat löydettävissä Somerville kirjasta. Somerville kirja eroaa huomattavasti toisesta seminaarin käytössä olleesta Mark G. Graff ja Kenneth R. van Wyk kirjasta ”Secure Coding: Principles & Practices”, jossa suurin osa esitetyistä asioista oli kirjoittajien omakohtaisten käytännön kokemusten esittämistä ilman tarkempaa viittausta tiedon syntyperään.

Tekstissä ei keskitytä varsinaisesti tietoturvallisen ohjelmiston laadintaan vaan yleensä turvallisen ja laadukkaan ohjelmiston laadintaa. Yhtenä syynä tähän on, että tietoturvallisen järjestelmän ohjelmointi on hyvin lähellä kriittisten järjestelmien rakentamisperiaatteita. Toisena syynä on, että Sommerville on ohittanut tietoturvallisuuteen liittyvät ohjelmistotuotannon seikat melko nopeasti ja teksti perustuu pääosin Sommervillen kirjaan. Kolmanneksi tietoturvallisuuteen keskittyviä ohjelmistoteknisiä tutkimuksia on tehty melko vähän verrattuna yleensä kriittisten järjestelmien ohjelmointiin.

Tämän osion tarkoituksena on johdattaa lukija kriittisten järjestelmien peruskäsitteisiin ja niihin liittyvin ohjelmistotuotannon menetelmiin. Lukijan tulee poimia itse seikat, joita voidaan hyödyntää tietoturvallisessa ohjelmoinnissa.

1.2. Kriittisistä järjestelmistä yleisesti

Kriittiset järjestelmät voidaan jaotella ensisijaisiin (primäärisiin) ja toissijaisiin (sekundäärisiin) kriittisiin järjestelmiin. Ensisijaiset kriittiset järjestelmät ovat järjestelmiä, joissa toimintahäiriö saattaa aiheuttaa vaaraa käyttäjille tai ympäristölle. Tällaisia ovat esimerkiksi lentokoneen ohjainjärjestelmä, jonka rikkoontuminen aiheuttaa todennäköisesti koneen tuhoutumisen ja vaaran koneessa ja maassa oleville. Toissijaiset eli sekundääriset kriittiset järjestelmät ovat järjestelmiä, joiden toimintahäiriö saattaa aiheuttaa vaara välillisesti tai epäsuorasti. Esimerkkinä vaikkapa lentokoneen mittarivalot, joiden toimimattomuus saattaa vaarantaa koneen ja matkustajien turvallisuuden laskeuduttaessa pimeällä. Kriittisten järjestelmien tärkeimpiä ominaisuuksia ovat

- luotettavuus (dependability)
- saatavuus (yleisesti myös käytettävyys) (availability)

- käyttövarmuus (reliability)
- käyttöturvallisuus (safety)
- tietoturvallisuus (security)

Näistä käyttövarmuus (reliability) on tärkein, koska ilman sitä muitakin ominaisuuksia on vaikea varmuudella saavuttaa

1.2.1. Saatavuus

Saatavuus (availability) tarkoittaa todennäköisyyttä, että järjestelmä on tiettynä ajan hetkenä toiminnassa ja pystyy tarjoamaan käyttäjän haluamat palvelut silloin, kun käyttäjä ne haluaa. Se ei välttämättä tarkoita, että järjestelmä pystyisi tarjoamaan ne palvelut, joita käyttäjä haluaa sen tarjoavan.

Suomessa yleisesti käytetty saatavuuden synonyymi on käytettävyys, joka tietoturvasanaston mukaan (Vahti, sanasto) tarkoittaa ominaisuutta, että tieto, tietojärjestelmä tai palvelu on siihen oikeutetuille saatavilla ja hyödynnettävissä haluttuna aikana ja vaaditulla tavalla.. Toisaalta se tarkoittaa helppokäyttöisyyttä. Tässä yhteydessä käytetään termiä saatavuus.

1.2.2. Käyttöturvallisuus (Safety)

Käyttöturvallisuus tarkoittaa järjestelmän kykyä toimia normaalisti tai epänormaalisti niin ettei se aiheuta vaaraa käyttäjille tai ympäristölleen. Vahdin sanaston (Vahti, sanasto) käyttöturvallisuus (operations security) tarkoittaa tietotekniikan käyttöön, käyttöympäristöön, tietojenkäsittelyyn ja sen jatkuvuuteen sekä tuki-, ylläpito-, kehittämis- ja huoltotoimintoihin liittyvä turvallisuutta. Määritelmät eivät ole yhtäpitävät, mutta tässä yhteydessä tarkoitetaa käyttöturvallisuudella edellä mainittua vaaran aiheuttamiseen liittyvää määritelmää.

Käyttöturvallisuuteen liittyy useita termejä, joista alla muutamia

- onnettomuus (accident) on suunnittelematon tapahtuma tai tapahtumasarja, joka aiheuttaa loukkaantumisen tai muun vahingon
- vaaratilanne (hazard) on olosuhde, joka voi johtaa onnettomuuteen
- vahinko tai vaurio (damage) on vahingon seurauksena syntynyt menetys
- vaaratilanteen merkityksellisyys (hazard severity) tarkoittaa suurimman mahdollisen vahingon määrä
- vaarantilanteen todennäköisyys (hazard propability) tarkoittaa vaaran todennäköisyyttä
- riski (risk) tarkoittaa todennäköisyys, että systeemi aiheuttaa vaaratilanteen

Käyttöturvallisuuden päätarkoitus on varmistaa, ettei synny vahinkoa tai ei aiheuteta vaaraa. Tämä tarkoittaa vaaran välttämistä (hazard avoidance), vaaratilanteen havaitsemista ja poistamista (hazard detection and removal) sekä vahingon rajoittamista (damage limitation).

Syitä, miksi käyttövarma ohjelmisto ei välttämättä ole käyttöturvallinen, on useita. Tyypillinen syy on, että vaatimusmärittely on puutteellinen. Myös laitteistovirhe saattaa aiheuttaa ohjelmiston odottamattoman toiminnan tai käyttäjä saattaa käyttää järjestelmää tavalla tai syöttää järjestelmään tietoja, jotka ovat sallittuja, mutta jotka juuri sillä hetkellä saattavat aiheuttaa epänormaalin toimintatilan.

1.2.3. Käyttövarmuus

Käyttövarmuus (reliability) tarkoittaa todennäköisyyttä, että järjestelmä tarjoa häiriöttömästi ne palvelut, joita sen on määritelty tarjoavan ja tämä tapahtuu määritetyssä ajassa määritetyssä käyttöympäristössä. Vahdin (Vahti, sanasto) mukaan reliability tarkoittaa luotettavuutta, joka on todennäköisyys, että tietojen tai tietojenkäsittelyprosessin eheys on säilynyt, tai että laite, järjestelmä tai palvelu toimii normaalisti tietyn ajan. Tämä onkin tyypillinen käännös ja määritelmä, mutta tässä yhteydessä se sekoittuisi toiseen luotettavuutta tarkoittavaan termiin, depenability, joka kuitenkin on tarkkaan ottaen parempi luotettavuuden määrittelemiseksi. Tässä yhteydessä käyttövarmuus tarkoittaa siis ensin mainittua määritelmää.

Käyttövarmuus sisältää myös saatavuuden koska, jos järjestelmä ei tarjoa niitä palveluita, joita sen on määritelty tarjoavan, se ei ole luotettava eikä täytä vaatimuksiaan. Käyttövarmuutta parantavia tapoja ovat mm.

- vikojen välttäminen (fault avoidance)
    - minimoidaan todennäköisyys erehdyksiin, ennen kuin ne johtavat vikaan
- vikojen havainnointi ja poisto
    - parannetaan verifioineilla ja validoinnilla mahdollisuutta havaita ja poistaa vika ennen kuin järjestelmä käytetään
- vikasietoisuus (fault tolerance)
    - varmistetaan, että järjestelmän viat eivät johda järjestelmä virheisiin eikä virheistä muodostu häiriöitä
        - häiriöllä (system failure) tarkoitetaan tiettynä hetkenä tapahtuvaa tilannetta, jolloin järjestelmä ei anna käyttäjän olettamia palveluita.
        - virheellä (system error) tarkoitetaan tilannetta, jolloin järjestelmä toimii määrittelyjen vastaisesti.

Vika (system fault) tarkoitetaan puolestaan järjestelmän virheellistä tilaa, johon sen ei vaatimusmäärittelyn ja edes suunnittelijoiden mukaan pitäisi mennä. Tämä voi kuitenkin tapahtua esimerkiksi inhimillisen virheen tai erehdyksen vuoksi (human error or mistake), jolloin kyseessä on inhimillisestä toiminnasta johtuva järjestelmän vika.

Järjestelmän käyttövarmuus on riippuvainen miten eri osatekijät on yhdistetty tai miten niiden toiminta liittyy toisiinsa. Järjestelmän, jonka osat A,B ,,,n, ovat toisistaan riippuvia, kokonais todennäköisyys virheelliseen toimintaan on

$$P_s = P_A + P_B + \ldots P_n$$

Tämä tarkoittaa, että kun osien lukumäärä kasvaa, todennäköisyys järjestelmähäiriöön kasvaa. Eli mitä monimutkaisempi järjestelmä on ja mitä enemmän on komponentteja ja osajärjestelmiä, sen todennäköisempää on järjestelmän virheellinen toiminta. Toisaalta järjestelmän, jonka osat (A,B ,,,n) ovat toisistaan riippumattomia, todennäköisyys virheelliseen toimintaan on

$$P_s = P_A * P_B * .... P_n$$

Tämä tarkoittaa, että järjestelmän yhden osan rikkoontuminen tai toimimattomuus ei vaikuta merkittävästi järjestelmän kokonaisturvallisuuteen, jos riippumattomia osajärjestelmiä on paljon. Tätä voidaan käyttää hyväksi esimerkiksi niin, että osaa P_A toisinnetaan (replicated) eli tehdään useita rinnakkaisia samanlaisia osia, jolloin P_A :n kokonaisturvallisuus P_a^n . Toisistaan riippumattomilla rinnakkaisilla järjestelmän osilla voidaan siis parantaa turvallisuutta.

Käyttövarmuuden parantaminen ohjelmistoissa on toisaalta helppoa, koska häiriöt ja virheet syntyvät yleisemmin niissä kohdissa järjestelmää (ohjelmaa), jota käytetään useimmin. Toisaalta käyttövarmuuden parantaminen on usein vaikeaa, koska kaikki ohjelmistoviat eivät ole aiheuta järjestelmävirheitä. Esimerkiksi eräässä ohjelmassa korjattiin 60% vioista, mutta se paransi vain 3% käyttövarmuutta. Järjestelmä voi siis sisältää paljon vikoja mutta voi olla (riittävän) käyttövarma.

1.2.3.1.Käyttövarmuuden metriikoita

Käyttövarmuutta voidaan mitata monella eri tavalla. Käyttövarmuuden metriikoita ovat esimerkiksi POFOD (Probability of failures on demand) ja sen arvona voi olla esimerkiksi 1: 1000. Mikä tarkoittaa todennäköisyyttä, että yksi tuhannesta toiminnosta tai tapahtumasta epäonnistuu. Toisin sanoen kuinka paljon on häiriöitä annettuja palveluja kohti. ROCOF (Rate of failure occurence) on hyvin samanlainen ja ilmoittaa kuinka monta epäonnistunutta tapahtumaa tai palvelua on tiettyä tapahtumamäärää kohti. MTTF (Mean time to failure) ilmoittaa keskimääräisen ajan häiriöiden välillä, esimerkiksi 1000h. Ajan mittauksessa voidaan käyttää kalenteriaikaa, käyntiaikaa toiminta-aikaa jne. AVAIL (availability) ilmoittaa kuinka suurella todennäköisyydellä järjestelmä on käytettävissä. Esimerkiksi 0,998 tarkoittaa, että järjestelmä on 2 tuntia tuhannesta pois käytöstä, muutoin käytettävissä. Kääntäen sanottuna sillä voidaan tarkoittaa mikä on järjestelmän toipumiseen kuluva aika (2h), suhteessa käyntiaikaan.

1.2.4. Ei-toiminnallisen käyttövarmuuden määrittely osana käyttövarmuutta

Ohjelmiston käyttövarmuus vaikuttaa yhtenä tekijänä järjestelmän kokonaiskäyttöturvallisuuteen. Muita tekijöitä ovat laitteiston käyttöturvallisuus sekä käytön käyttöturvallisuudesta. Tämä tarkoittaa, että vaatimusmäärittelyssä tulee olla toiminnalliset määrittelyjen lisäksi (miten järjestelmän pitää toimia) myös selvitys ei-toiminnallista vaatimuksista, kuten käytettävyydestä ja käyttöturvallisuudesta. Turvallisuuden kannalta ei-toiminnalliset vaatimukset tarkoittavat usein määrittelyjä miten järjestelmä ei saa toimia.

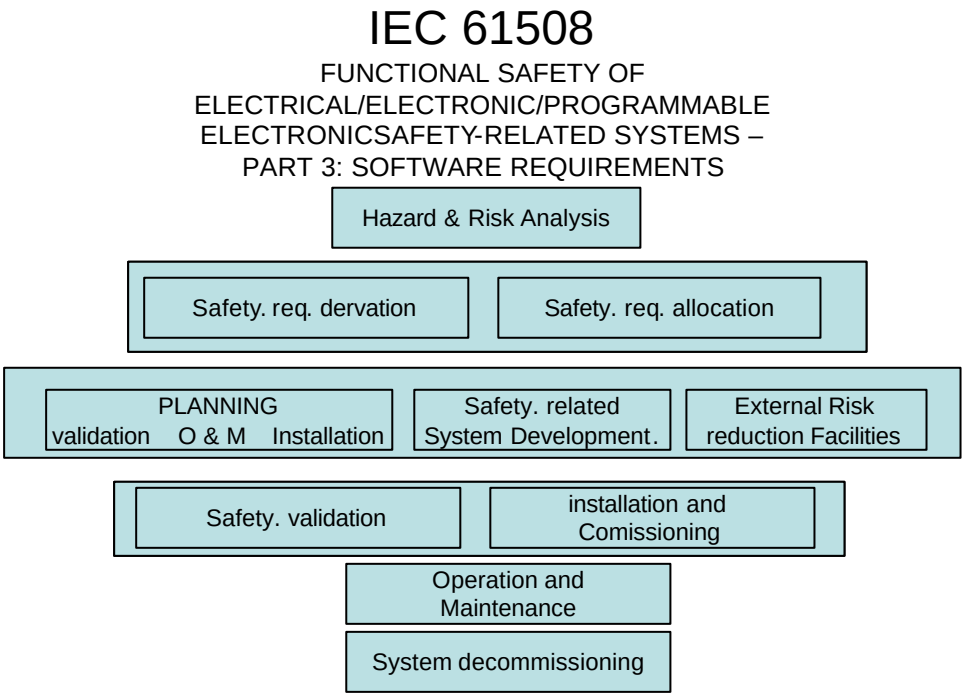
Ei-toiminnalliset määrittelyjen laadinnassa pitää ottaa huomioon, että subjektiiviset näennäismäärittelyt kuten ”vikoja ei saa olla kuin max 1kpl 1000 koodiriviä kohti”, ovat turhia, koska ei pystytä määrittelemään miten viat löydetään. Toiseksi em. määrittely ei kerro järjestelmän turvallisuudesta mitään. Ei-toiminnallisten määrittelyjen pitäisi olla konkreettisia, järjestelmän turvallisuutta mittaavia tai sitä konkreettisesti parantavia.

Ei-toiminnallisissa määrittelyssä voidaan käyttää hyväksi erilaisia häiriöiden luokittelua (failure classification). Häiriöt voidaan luokitella esimerkiksi

- tilapäinen häiriö (transient), joka saattaa tulla vain tietyillä syötteillä tai satunnaisilla käyttötilanteilla
- pysyvä häiriö (permanent) sattuu kaikilla syötteillä
- toivuttavissa oleva (recoverable) häiriö tarkoittaa, että normaali toiminta voidaan palauttaa ilman käyttäjän toimintaa
- ei-toivuttavassa häiriössä (unrecoverable) toipuminen vaatii erityistoimia

Lisäksi voidaan luokitella, miten häiriö vaikuttaa järjestelmän stabiilisuuteen tulevaisuudessa. Ei-korruptoiva (non-corrupting) häiriö ei korruptoi dataa tai järjestelmän tilaa ja järjestelmä pysyy stabiilina. Korruptoiva (corrupting) häiriö korruptoi järjestelmän tilan tai data korruptoituu, jolloin järjestelmä ei välttämättä ole enää stabiili. Tällöin sen toiminnan tila tulee palauttaa johonkin tunnettuun stabiiliin tilaan.

Kriittisen järjestelmän ei-toiminnallisessa määrittelyssä analysoidaan jokainen alijärjestelmä ja tutkitaan mitä häiriötä niissä saattaa olla ja mitä vaaroja ne aiheuttavat. Häiriöt voidaan luokitella edellä mainitulla tavalla ja jokaiseen häiriöluokkaan valitaan sopiva luotettavuusmetriikka (POFOD), ROFO /(ROCOF), MTTF, AVAIL), jonka perusteella saadaan konkreettinen vaatimus turvallisuudesta. Lisäksi, silloin kun on mahdollista, määritellään toiminnalliset käyttöturvallisuusvaatimukset, jotka määrittelevät järjestelmän toiminnan häiriöiden minimoimiseksi



Kuva 1. Toiminnallinen määrittely IEC 61508 mukaisesti (IEC, 1998).

Vaaratilanteiden ja riskin arviointi (Hazard & Risk analysis & techniques) on iteratiivinen prosessi, jossa tulee jokaisella kierroksella tehdä seuraavat toimenpiteet

1. vaaran tunnistaminen

2. riskin arviointi ja vaaran luokittelu

3. vaaran jakaminen osiin

4. riskin rajoittamisen arvioiminen

Tekniikoita on useita kuten esimerkiksi vikapuut, tarkistuslistat, petriverkot, jne. Hyvin perusteellinen vaaratilanteiden ja riskikartoitustekniikoiden selvitys löytyy Nancy Levesen kirjasta ”Safeware, System Safety and Computers” (1995).

Kun vaarat on analysoitu, arvioidaan niistä johtuvat riskit. Riskit voidaan luokitella

- ei-hyväksyttäviin (intoleranble) riskeihin, joita ei voida hyväksyä missään olosuhteissa

- alhaisin mahdollinen riski käytännössä (As low as reasonably practical, ALARP), joka voidaan hyväksyä, jos on riittävät riskin alentamismahdollisuudet

- hyväksyttävät merkityksettömät riskit, joilla ei ole olennaista merkitystä turvallisuuteen

Ei-hyväksyttävien riskien estämiseksi pitää tehdä kaikki voitava ja ne tulisi saada muutettua joko ALARP –tasoiseksi tai täysin mitättömiksi.

1.3. Tietoturvallisuus

Järjestelmän turvallisuus (security) on arvio järjestelmän kyvystä suojautua ulkopuolista, joko vahingossa tai tarkoituksella tapahtuvaa hyökkäystä vastaan. Se tarkoittaa havoittuvuuden välttämistä (vulnerability avoidance), hyökkäyksen havainnointia ja kumoamista (attack detection and neutralization), vaaralle alttiina olon rajoittamista (Exposure limitation). Vahdin (Vahti ,sanasto) mukaan security tarkoittaa turvallisuutta, joka on olotila, jossa tiedossa olevat uhat eivät merkitse sanottavaa riskiä. Tässä yhteydessä security tarkoittaa tietoturvallisuutta erona käyttöturvallisuudesta (safety).

Tietoturvallisuuden ja käyttöturvallisuuden määritys sisältää samoja asioita ja ei-toiminnalliset vaatimukset (shall not…) ovat molemmissa tyypillisiä. Suurimmat erot ovat

- tietoturvallisuuden määrittelyissä ei useinkaan pystytä määrittämään tietoturvallisuuden elinkaarta kokonaisuudessaan sillä tavoin kuin käyttöturvallisuudessa

- suuri osa tietoturvallisuuden ei-toiminnallisista vaatimuksista on yleispäteviä kun taas käyttöturvallisuudessa vaatimukset ovat tapauskohtaisia

- osa tietoturvallisuuden osa-alueista on kypsiä (kryptosysteemit, -protokollat, autentikointi jne.) mutta turvamekanismien siirtäminen yleiseen käyttöön ontuu

1.4. Järjestelmän selviytymiskyky ja poikkeusten hallinta

Selviytymiskyky (survailability) uhkaavia tekijöitä vastaan on turvallisen järjestelmän tärkeimpiä ominaisuuksia. Se tarkoittaa, että järjestelmän tulee olla vikasietoinen ja sen tulee pystyä säilyttämään turvallinen stabiili toimintatila tai toimimaan tilanteen edellyttämällä poikkeavalla tavalla myös hyökkäyksen aikana.

Poikkeus sattuu kun järjestelmässä tapahtuu jotain odottamatonta. Poikkeuksia voidaan hallita seuraamalla laajamittaisesti kaikkia mahdollisia poikkeustilanteita ja toimitaan hallitusti niistä poispääsemiseksi. Se valitettavasti lisää ohjelman monimutkaisuutta ja vika-alttiutta sekä heikentää ohjelman ymmärrettävyyttä. Toisena vaihtoehtona on, että palautetaan tilanne edelliselle kutsuvalle ohjelmanosalle, jolla vikaa ei vielä ollut. Myös se lisää ohjelman monimutkaisuutta kuten edellä. Poikkeusten hallinta voidaan toteuttaa myös ohjelmointikielen antamalla tuella.

1.5. Vikasietoisuus

Vikasietoisuus (fault tolerance) tarkoittaa, että järjestelmä pysyy toiminnassa vaikka siinä esiintyisikin vikoja. Tärkeimmät vikasietoisuuden menetelmät ovat

- vikojen havainnointi ja mahdollisten virheellisten tilakombinaatioiden syntymisen estäminen
- vahinkojen arviointi ja rajaus niin, että vika pystytään rajaamaan tiettyyn ohjelmiston osaan
- vioista toipuminen, jolloin järjestelmä palauttaa itsensä turvalliseen tilaan
- vikojen korjaus ja havaittujen vikojen korjaaminen niin ettei sellaista enää satu

Vikaisietoisuuden lähestymistavat ovat

- defenssivinen, puolustava ohjelmointi, jossa oletetaan, että havaitsemattomia vikoja on olemassa ja tarkkaillaan, että tilasiirtymät yms. ovat aina sallituja ja oikeita
- vikasietoinen arkkitehtuuri, jossa arkkitehtuuri estää vikatilanteiden synnyn. Haittana on, että sen edellyttää laitteiston ja ohjelmiston selkeää tukea

1.5.1.1. Vikojen havainnointi ja rajaus

Vika tulisi havainnoida joko silloin kun se tapahtuu tai jo ennen kun viallinen ohjelmiston osa suoritetaan. Ennaltaehkäisevässä tavassa vian havainnointi alustetaan ennen tilasiirtymää ja jos vika esiintyy, virheellistä tilasiirtymää ei tehdä. Tämä edellyttää, että ylläpidetään tilatietoa (if (ok)….). Jälkikäteisessä havainnoinissa vianhavainnointi alustetaan vasta, kun tilasiirtymä on tehty ja jos vika havaitaan, laukaistaan toipumismenettely: ok = ReadFile (…); if (ok) …. else..

Vikojen rajauksen tarkoituksena havainnoida ja rajata ohjelmistosta ne osat, jotka ovat vaurioituneet vian vuoksi (vrt korruptoiva häiriö). Tähän voidaan käyttää tarkistuslippua tms., joka tarkastetaan ennen ohjelmiston osien ja tilasiirtymien suorittamista.

1.5.1.2. Viasta toipuminen

Kun vika on havaittu ja arvioitu, pyritään toipumaan vahingosta. Eteenpäin suuntautuvassa toipumisessa pyritään korjaamaan ohjelman tilan normaaliksi. Tämä voidaan tehdä esimerkiksi silloin, kun data on korruptoitunut tai kun linkitetyt tietorakenteet ovat korruptoituneet (vaatii kahden suuntaiset osoittimet).

Takautuvassa toipumisessa palautetaan jonkin turvalliseksi tiedetyn tilan. Poikkeusten hallinta on tyypillinen tämän tyyppinen menettely.

1.5.1.3. Vikasietoiset arkkitehtuurit

Defensiivinen ohjelmointi on tehokas tapa vikasietoisuuden lisäämiseksi Se on yksinkertainen eikä lisää monimutkaisuutta mutta se ei kykene vastaamaan tilanteisiin, jotka johtuvat laitteiston ja ohjelmiston rajapinnassa olevista vioista. Tyypilliset vikasietoiset arkkitehtuurit ovat Triple-modular redundancy (TMR), n- versio-ohjelmointi ja Recovery blocks

Triple-modular redundancy (TMR)on arkkitehtuuri, jossa laitteisto on monennettu ja jokaisen ulostuloa verrataan. Jos jonkin ulostulo poikkeaa, se hylätään. Siinä oletetaan, että yksittäinen komponentti vioittuu, eikä oleta suunnitteluvirhettä. Käytännössä se tarkoittaa, että samasta vaatimusmäärittelystä tehdään eri suunnittelijoiden osajärjestelmät, joita käytetään rinnakkain.

N- versio ohjelmoinnissa samasta speksistä ohjelmoidaan eri suunnittelijoiden ohjelmanosat, joita käytetään rinnakkain eri koneissa. Versioita ajetaan rinnakkain ja oikea lopputulos ratkaistaan esimerkiksi äänestämällä. Vaatimuksena on, että versioita on useita ($n > 3$). Tämä tekee järjestelmästä monimutkaisen.

Recovery blocks arkkitehtuurissa jokainen ohjelmistokomponentti sisältää oikeellisuuden tarkistuksen sekä toisen vastaavan ohjelmistolohkon mikäli virhe havaitaan. Versioita ajetaan peräkkäin. Sekä ensisijainen että toissijainen ohjelmistolohko on implementoitu samasta speksistä erillisesti mutta sekin tekee järjestelmästä monimutkaisen.

Eri suunnittelijoiden ja ohjelmoijien erilaisuus tarkoittaa myös todennäköisyyttä erilaisiin virheisiin eri kohdissa ohjelmaa. Tätä voidaan käyttää hyväksi vikasietoisuudessa. Voidaan esimerkiksi tehdä implentoinnit eri kielillä, käytetään eri ohjelmointiympäristöjä ja -välineitä tai käytetään eri algoritmeja. On kuitenkin huomioitava, että menettely olettaa, että vaatimusmäärittely on oikein tehty!

1.6. Kriittisten järjestelmien kehittäminen ja laadukas ohjelmistotuotanto

Turvallisen systeemin suunnittelun lähtökohtana on, että järjestelmä pidetään niin yksinkertaisena kuin mahdollista ja kriittiset osat eristettään muista osista. Etuna tässä on, että kriittisiä osia on todennäköisesti vähän, jolloin ne voidaan helpommin tehdä korkealaatuisiksi.

Kriittisen järjestelmän kehittämisessä on kaksi lähestymistapaa

- vikojen välttäminen (fault avoidance)

    - vikojen havainnointi (fault detection)
    - vikojen minimointi (fault minimisation)
- vikasietoisuus (fault tolerance)

Vikojen minimointi tulisi olla aina ohjelmistokehityksen tavoitteena. Vaikka ohjelmassa ei olisi vikoja, se voi silti aiheuttaa häiriöitä. Koska vika on vaatimusmäärittelyn vastainen toiminta, tulee vaatimusmäärittely olla tarkka ja mieluummin formaali. Myös ohjelmistotuotannon laatujärjestelmä ja laatukulttuuri pitää olla olemassa ja ohjelmistoprosessi tulee olla määritetty ja opetettu kaikille. Luotettavassa ohjelmistoprosessissa tulee

- tarkastaa vaatimusmääritykset ja vaatimusmäärittelyiden johtaminen ja niiden muutostenhallinta
- tarkastaa käytetyt mallit (tila-, luokka, jne)
    - kaikkien yksittäisten mallien tulee olla sisäisesti ristiriidattomia
    - kaikkien mallien tulee ole toistensa suhteen ristiriidattomia
- tarkastaa suunnittelu ja koodi staattisin analyysein
- suunnitella ja johtaa testien suoritus

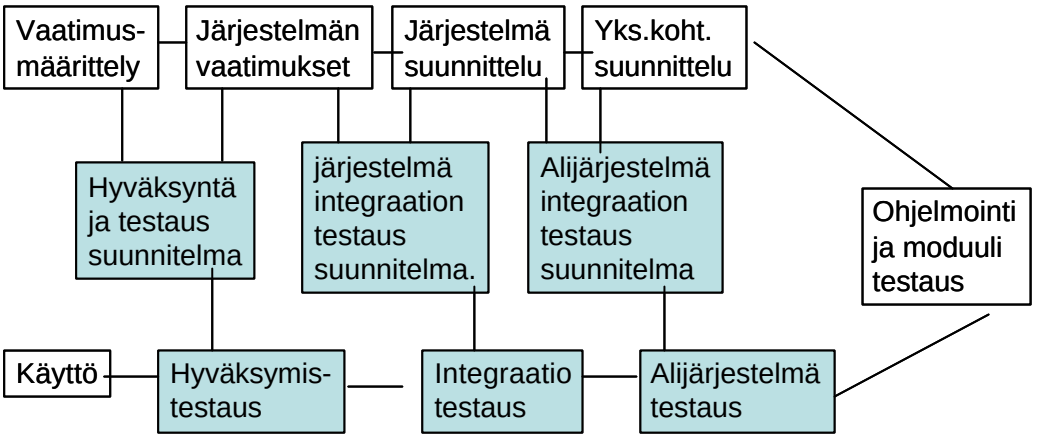
Virheiden välttämisessä (error avoidance) tulee ottaa huomioon, että ohjelmoijat ovat ihmisiä ja ihmiset tekevät virheitä. Tämän vuoksi tulee käyttää sellaisia ohjelmistotekniikoita, ohjelmistorakenteita ja välineitä, että virheiden määrä olisi mahdollisimman pieni. Käytännössä tämä tarkoittaa, että tulisi välttää esimerkiksi seuraavia ohjelmistorakenteita

- pointterit (vaarana virheelliset viittaukset)
- liukuluvut (mahdoton vertailla ilman erityistoimenpiteitä)
- rinnakkaisuus (ajoitusongelmat), säikeiden ja tehtävien (task) käyttö suotavampaa
- rekursiot (logiikan seuraaminen vaikeutuu)
- keskeytykset (kontrollin siirtäminen pois ohjelmasta)
- perintä (käyttäytymisen seuraaminen ja analysointi vaikeutuu)
- aliaksien käyttö (väärät viittaukset)
- oletus input:in käyttö (avoimia hyökkäämiselle)
- muistin varaus (muisti tulee varata ja vapauttaa ajonaikana, ei käännöksen aikana)

Tiedon jaossa tulisi aina toteuttaa ”need to know” periaatetta ja varsinaiset implementoinnit tulisi kätkeä rajapinnan taakse

1.7. Ohjelmistojen verifiointi ja validointi

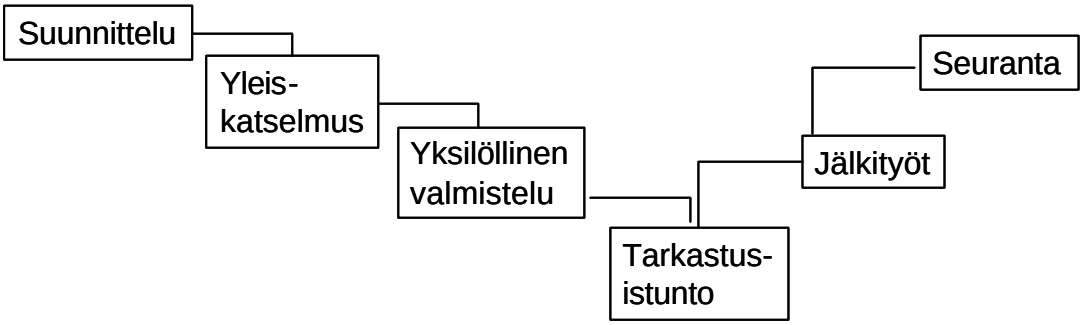
Validointi tarkoittaa olemmeko tekemässä oikeaa tuotetta ja täyttääkö ohjelmisto sille asetetut vaatimukset. Verifiointi puolestaan tarkoittaa teemmekö tuotteen oikein ja onko ohjelmisto vaatimusten mukainen. Verifiointi ja validointi (V&V) on koko elinkaaren mittainen prosessi ja sen suoritustapoja ovat tarkastukset ja testaukset.



Kuva 2. Verifioinnin ja validoinnin V-malli (vaakatasoon käännettynä)

1.7.1. Ohjelmiston tarkastukset

Tarkastuksissa tarkastetaan, että vaatimusmäärittelyt, dokumentit, ohjelmistot ja testaukset on oikein tehty. Sitä voidaan ja tulee tehdä jatkuvasti.



Kuva 3. Ohjelmiston tarkastuksen toimenpiteet

Tyypillisiä tarkastuksissa havaittuja virheitä ovat data-virheet, ohjelmiston kontrollivirheet, input/output-virheet, rajapintavirheet, tallennusvirheet ja poikkeustenhallinnan virheet.

Tarkastuksia voidaan myös automatisoida. Niissä pyritään tyypillisesti analysoimaan seuraavat seikat.

- ohjelmistokontrollin siirtymiset
- datan käyttö
- rajapinnat
- tietovirrat

- ohjelmiston suorituspolut

Ohjelmistokontrollin siirtymisessä analysoidaan miten ohjelmiston kontrolli siirtyy ohjelman suorituksen edetessä. Tyypillinen virheitä aiheuttava kontrollirakenne on silmukka, joissa on monia poistumisia samoin kuin useita sisäkkäisiä ehtolauseita sisältävät rakenteet. Myös käyttämätön koodi, jonne ohjelmiston suoritus ei milloinkaan siirry, pyritään löytämään.

Datan käytön analyysissä tarkastetaan muuttujien käyttöä tai niiden käyttämättömyyttä eli turhia muuttujien olemassa oloa, muuttujien alustuksia, viittauksia ja samannimisyyksiä. Tarkoituksena on löytää ohjelmiston suorituksen aikaisen datan tallennukseen, käyttöön muuttamiseen ja poistoon liittyvät virheet.

Rajapinta-analyysissä tarkastetaan funktio- ja proseduurikutsut, niissä välitettävät parametrit ja parametrien käyttö tai käyttämättömyys. Virheellisen parametrien käytön määrittely on hyvin vaikeaa, siksi ohjelmisto tulisi rakentaa sellaiseksi, ettei virheellinen parametrin käyttö ole mahdollista.

Tietovirtojen analyysissä tarkastetaan inputin ja outputin vastaavuudet ja riippuvuudet. Polkuanalyysissä pyritään tarkastamaan kaikki mahdolliset suorituspolut. Käytännössä tämä ei aina ole mahdollista.

Niin sanottu ”Cleanroom- ohjelmistonkehitys” on tiukkaan tarkastusprosessiin perustuva ohjelmiston kehitystapa. Siinä ohjelmistokomponenttien-testaus on korvattu tarkastuksilla ja siinä pyritään laatimaan ohjelmia, joissa ei ole ollenkaan virheitä. Se vaatii formaalien määrittelyjen käyttöä ja inkrementaalista ohjelmistonkehitystä, strukturoitua ohjelmointia ja sen staattista verifiointia sekä staattista järjestelmän testausta.

1.7.2. Ohjelmiston testaus

Testaukset ovat dynaamisia, valmiin tuotteen tai osan toiminnallista testaamista oikealla datalla, oikeassa ympäristössä. Sitä voidaan tehdä vasta kun proto, ohjelmisto tai sen osa on valmis. Käytännössä testaus perustuu aina intuitioon miten ohjelman pitäisi toimia.

Testaus on virheiden ja puutteiden etsintää, jossa testataan vastaako ohjelmisto vaatimusmäärittelyä. Ohjelma pyritään testauksessa samaan tilaan, jossa se toimisi vaatimusmäärittelyjen vastaisesti. Mikäli onnistutaan näin tekemään, voidaan osoittaa virheen olemassaolo. Sen sijaan testauksella ei voida osoittaa virheiden puuttumista.

Toiminnallisessa eli ”Black Box” –testauksessa testaussuunnitelma johdetaan ohjelmistomäärittelystä. Siinä järjestelmä on ”mustalaatikko”, joka toimintaa voidaan seurata inputin ja outputin perusteella ja jos ulostulo ei ole odotettu, testi on onnistunut. Testaus on onnistuneesti havainnut virheen. Black-box testauksen ongelmana syötejoukon muodostaminen.

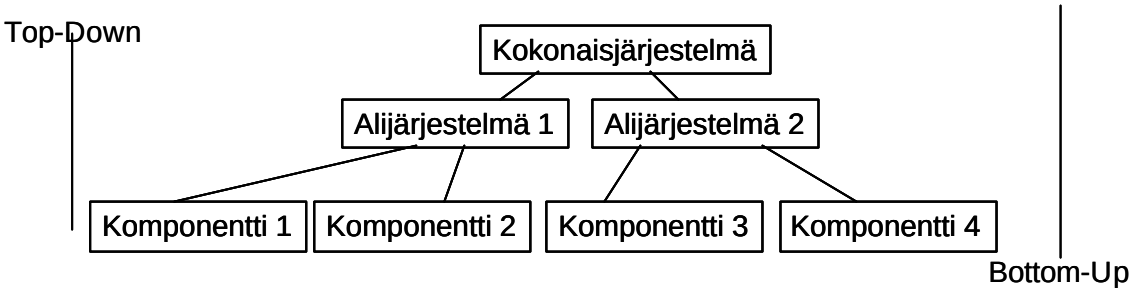
Rakenteellisessa eli ”White-box” testauksessa testaus johdetaan ohjelmiston rakenteesta ja sitä käytetään usein pienissä ohjelmiston osissa. Tyypillisesti testattavan koodin tai algoritmin rakenteesta voidaan johtaa testauksen syöteavaruus.

Polkutestaus on rakenteellisen testauksen eräs tyyppi, jossa testataan ohjelmistokomponentin erilliset suorituspolut. Kaikkien polkujen testaaminen on kuitenkin mahdoton-

ta. Erityisesti olio-ohjelmoinnin polymorfisuus ja dynaaminen perintä tekee sen mahdottomaksi.

Kun komponentit on testattu, voidaan tehdä integraatiotestaus. On muistettava, että koska virheen syyn löytäminen on vaikeaa, on syytä käyttää inkrementaalista testausta, jossa aloitetaan minimaalisesta systeemistä ja lisätään testattavaa systeemiä pikkuhiljaa.

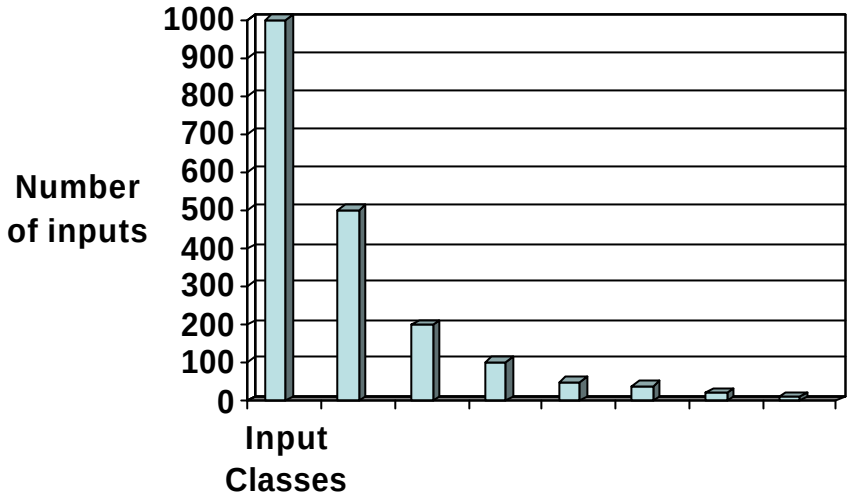
Integraatiotestauksen suorittamisessa on kaksi lähestymistapaa. Top-down testauksessa integroidaan ensin korkean tason komponentin ja testataan ne ja sitten siirrytään alemman tason komponentteihin. Top-down testaus sopii hyvin ohjelmistokehityksen alussa tapahtuvana arkkitehtuuritestaukseen. Haittana on että alimpia osia ei vielä ole ja niiden toimintaa pitää simuloida. Bottom-up testauksessa integroidaan alemman tason komponentit ja testataan niiden toiminta. Se sopii yksittäisten ohjelmistokomponenttien ja pienten osajärjestelmien testaukseen jo siinä vaiheessa toteutusta, kun kokonaisjärjestelmää ei vielä ole. Haittana on, että testattavia komponentteja käyttävät ohjelmiston osat ja niiden toiminta tulee simuloida eli laatia komponenttien testausympäristö.



Kuva 4. Ohjelmiston testaus ja sen lähestymistavat

Varsinaisten komponenttien ja osajärjestelmien testauksen lisäksi voidaan tehdä rajapintatestausta, jossa testataan parametrien ja viestien välitys, jaettujen resurssien, kuten muistin käyttö. Rajapintatestaus on erittäin tärkeää erityisesti olio-ohjelmissa. Kun järjestelmä tai sen osa on kokonaisuudessaan testattu, voidaan vielä tehdä kuormitustestaus, jossa testataan miten nopeasti ja luotettavasti järjestelmä kykenee vastaamaan oletettuun maksimikuormaan.

Testauksessa käytetyn syötejoukon muodostamisessa auttaa hyvä sovellusalueen tunteminen. Jos syöteavaruus on suuri, voidaan kaikkien arvojen testauksen sijasta käyttää syötejoukon vastaavuusluokkia ja niistä erityisestä keskimääräisiä ja vastaavuusluokan päätearvoja. Idea perustuu siihen, että usein ohjelmat reagoivat samankaltaisiin syötteisiin samalla tavalla. Tällöin testejä voidaan yksinkertaistaa ja nopeuttaa muodostamalla kuvitelluista syötejoukosta ominaisuustyypit ja niitä vastaavat ulostulot ja ajetaan testit niillä. Vastaavuusluokkia vastaavien syötearvojen käytön tiheys tullee ottaa huomioon testeissä eli useimmin käytettyjen vastaavuusluokkien arvoilla tulee tehdä laajemmat testit kuin vähemmän käytetyillä. Mikäli järjestelmä on täysin uusi, ei syöteavaruuden käyttötiheydestä ole varmuutta, vaan ne pitää olettaa tai testata kaikilla vastaavuusluokilla samalla tavalla.



Kuva 5. Syöteavaruuden vastaavuusluokkien tiheyksiä.

1.8. Kriitisen järjestelmän validointi

Kriittisten järjestelmien kehittäminen on periaatteessa samanlaista kuin muidenkin järjestelmien mutta mahdollisen vahingon suuruus ja kustannukset vaativat tarkemman validoinnin.

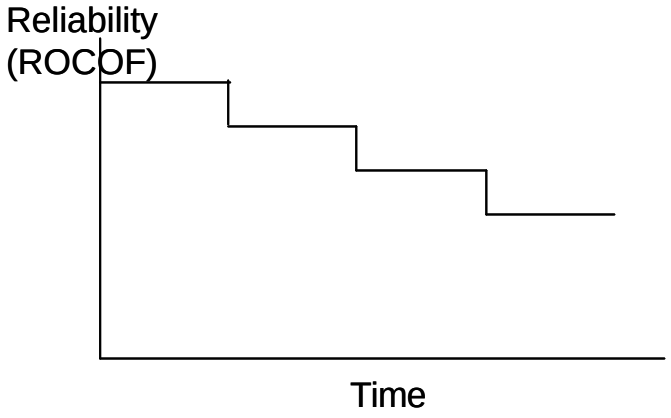
Tyypillisesti validointi perustuu formaalien menetelmien käyttöön. Niiden haittana on kuitenkin se, että sovellusalueen asiantuntijalla ei välttämättä ole ohjelmistotekniikassa käytettyjen formaalien menetelmien osaamista. Tällöin kommunikointi ohjelmistosuunnittelijoiden ja sovellusalueasiantuntijoiden välillä voi olla ongelmallista. Myös kustannusten epälineaarisuus on ongelma. Järjestelmän monimutkaisuuden lisääntyminen ei kasvata validoinnin kustannuksia lineaarisesti vaan huomattavasti tätä enemmän. Toisaalta formaalilla menetelmällä tehty verifiointi ja validointi ei takaa, että järjestelmässä ei olisi vikoja. Haitoista huolimatta formaalit menetelmät ovat käyttökelpoisia monessa sovellusalueessa vaikka eräillä muilla järjestelmillä saadaan parempi kustannustehokkuus.

1.8.1. Luotettavuuden validointi

Validoinnissa voidaan käyttää hyväksi olemassa olevasta järjestelmästä saatuja tietoja ja uuden järjestelmän ”tilastollinen testauksen” tuloksia. Tämä edellyttää samankaltaisen olemassa olevan järjestelmän käyttöprofiilien opettelua ja testausdatan konstruointia sekä testausta em. aineistolla käyttämällä sopivia metriikoita. Kun riittävä määrä testiaineistoa ja virheitä on saatu, voidaan laskea järjestelmän luotettavuus. Tilastollisen luotettavuuden validoinnilla on vaikeutena käyttöprofiilien virheet ja epävarmuus, korkeat testauskustannukset ja heikko tilastollinen näyttö. Luotettavuus on perusominaisuuksiltaan subjektiivinen asia.

Koska testaus on kallista, se tulee lopettaa heti, kun riittävä luotettavuustaso saavutetaan tai jos huomataan, ettei sitä voida saavuttaa. Luotettavuuden oletettuun kehittymiseen on kehitetty erilaisia malleja. Askelmallin (Step function model) oletuksena on, että luotettavuus kasvaa joka korjauksen jälkeen lineaarisesti ja jokainen korjaus poistaa

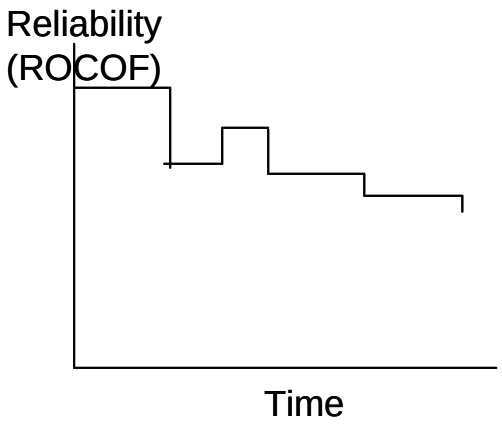
virheen eikä luo uusia virheitä. Tällöin virheiden määrä (ROCOF) alenee lineaarisesti ja vastaavasti luotettavuus kasvaa lineaarisesti jokaisella korjauskerralla.



Kuva 6. Luettavuuden askelmallin periaate

Askelmallin on erittäin yksinkertaistettu malli todellisuudesta eikä luotettavuuden muuttuminen käytännössä muutu mallin osoittamalla tavalla. Käytännössä virheen korjaus saattaa tuoda uusia virheitä eikä luotettavuus ei aina parane virheen korjauksen yhteydessä. Näin ollen virheiden korjaaminen ei paranna luotettavuutta lineaarisesti vaan jotkin virheet voivat heikentää luotettavuutta ja jotkut parantaa sitä huomattavasti. Käytännössä pahimmat virheet löytyvät ensin, jolloin aluksi luotettavuus kasvaa voimakkaammin. Toisaalta usein oletettu luotettavuus on parempi kuin todellinen.

Askelmallista kehittyneempi versio on satunnaisaskelmalli, jossa oletuksena on, että luotettavuus ei kasvaa joka korjauksen jälkeen lineaarisesti mutta muuttuu kuitenkin diskreetisti. Se sallii aluksi suurimmat muutokset ja ROCOF voi muuttua korjauksessa kumpaan suuntaan tahansa.



Kuva 7. Satunnaisaskelmallin periaate

Luotettavuusmallien käytöllä saavutettuja etuja on, että testauksen suunnittelu helpottuu, koska voidaan arvioida testaukseen kuluva aika. Myös tilaajalle tai asiakkaalle on helpompi esittää luotettavuuden muuttuminen ja kustannusten kasvaminen ja on helpompi löytää optimitilanne kustannusten ja luettavuuden välillä.

1.8.2. Käyttöturvallisuuden varmistaminen

Käyttöturvallisuuden varmistaminen eroaa luottavuuden varmistamisesta monella tapaa. Ensinnäkin käyttöturvallisuutta ei voida aina määritellä kvantitatiivisesti eikä sen vuoksi testata tai todentaa kvantitatiivisesti. Tämä tarkoittaa, että joudutaan subjektiivisiin määrityksiin.

Käyttöturvallisuuden verifiointi ja validointi on lähes samanlaista kuin V&V yleensäkin pl. tilastolliset menetelmät eivät aina sovi, koska ns hyvin pienien/suurien lukujen matematiikka tulee ongelmaksi. Käyttöturvallisuudessa laaja-alainen tarkastus on olennaista ja erityisesti suunniteltujen toimintojen tarkastus, niiden ylläpidettävyys ja ymmärrettävyys. Olennaisempaa on löytää virheet, jotka johtavat vaaratilanteeseen kuin poistaa kaikki järjestelmässä olevat virheet

Käyttöturvallisuuden parantamiseksi ja järjestelmän oikeellisuuden todentamiseksi kannattaa määritellä turvallisuuden argumentit ja tekijät, joiden avulla todennetaan järjestelmän oikea toiminta. Tällöin ei tarvitse todistaa, että koko ohjelma toimii oikein vaan ainoastaan turvallisuuteen liittyvät asiat. Turvallisuusargumentit ovat helpompi ja nopeampitapa todentaa oikea toiminta kuin formaali oikeaksi todistaminen. Turvallisuusargumentit saadaan uhka-analyysistä ja jokainen uhka ja siihen johtavat ohjelman toiminnot ja ohjelman osat analysoidaan ja tarkastetaan. Nämä poistetaan tai korjataan ohjelmasta kaikki vaaraan johtavat polut. Lopuksi todennetaan, että vaara ei voi syntyä.

1.8.3. Tietoturvallisuuden arviointi

Tietoturvallisuuden arviointi on tullut yhä tärkeämmäksi osaksi ohjelmistonkehitystä. Se on samantyyppistä käyttöturvallisuuden kanssa pl. käyttöturvallisuuden vahingot yleensä tahattomia ja tietoturvallisuudessa tahallisia. Toisaalta käyttöturvallista järjestelmää voi käyttää väärin tietoturvallisuuden rikkomisessa tai hyökätä sitä vastaan. Myös pitkään turvallisena pidetty järjestelmä voi joutua hyökkäyksen kohteeksi.

Lähteet

IEC 61508, International Electrotechnical Commission standard. IEC, Geneva, Switzerland ,1998.

Nancy G. Leveson. Safeware, System Safety and Computers. Addison –Wesley Publishing Comppany Inc. 1995.

Ian Sommerville, Software Engineering, 6th ed.,Addison-Wesley Publisng Inc. 2001

Vahti, Hallinnon kehittäminen, Tietoturvasanasto
http://www.vn.fi/vm/kehittaminen/tietoturvallisuus/vahti/sanasto/

2. Tietoturva ohjelmistojen suunnittelussa

Heikki Vatiainen

<hessu@cs.tut.fi>

Tampereen teknillinen yliopisto

Tietoliikennetekniikan laitos

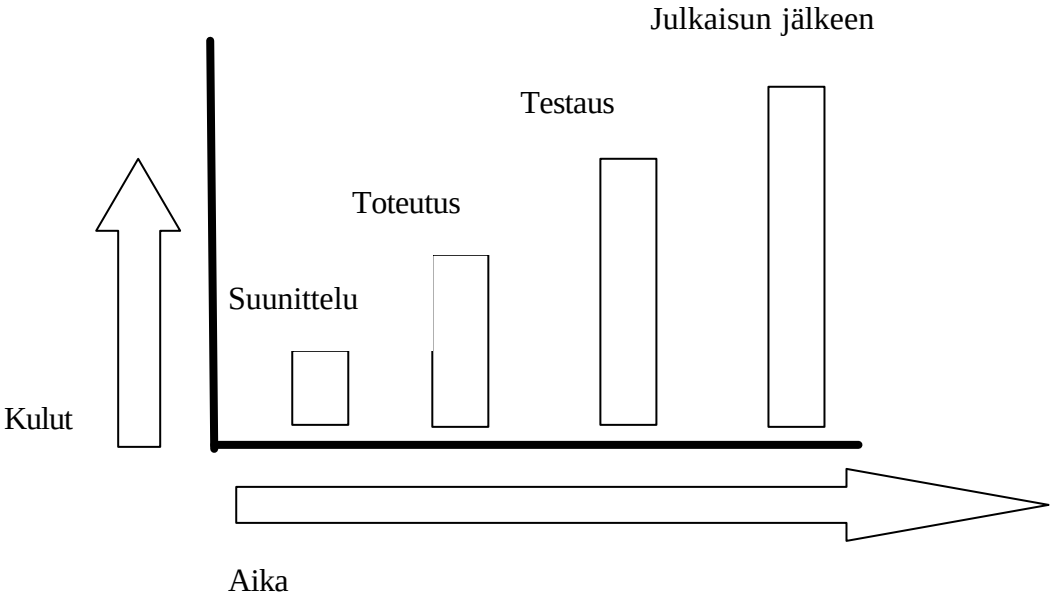
2.1. Johdanto

Tämä teksti koostaa kirjoittajien Mark G. Graff ja Kenneth R. van Wyk teoksen Secure Coding: Principles & Practices luvussa 3 “Design” esitettyjä ideoita ja ajatuksia. Kirjan asioiden tiivistämisen lisäksi olen ottanut mukaan joitain omia ajatuksiani, jotka heräsivät kirjan tekstin pohjalta sekä yhden esimerkin, joka sattui tämän kurssin aikana.

Vaikka tämä teksti käsittelee vain yhtä lukua, ei turvallinen ohjelmointi rajoitu vain turvalliseen suunnitteluun. Turvallinen suunnittelu on osa holistista lähestymistapaa, jossa turvalliset arkkitehtuuri, suunnittelu, toteutus ja testaus sekä vielä julkaisun jälkeinen käyttö ovat yksi kokonaisuus, joilla turvallinen ohjelmointi saadaan toteutettua.

2.2. Miksi suunnitella tietoturvallisesti?

Tietoturvaongelmien selvittäminen suunnitteluvaiheessa on sekä taloudellisesti että toteutuksen kannalta kaikista kannattavinta. Syyksi tähän Graff ja van Wyk osoittavat kustannuksien ja työmäärän kasvamisen siirryttäessä suunnittelusta toteutukseen ja siitä edelleen testaukseen ja tuotteen julkaisun jälkeiseen tukeen. Kuva 1 esittää ajan ja kustannusten suhdetta toisiinsa. [GRAFF]



Kuva 8. Tietoturvavirheiden korjaamisen hinta elinkaaren eri vaiheissa

Suunnitteluvaihe ohjelmien ja ohjelmistojen tuotannossa luo perustan ohjelmiston tulevalle ylläpidolle ja lähes väistämättä tulevaisuudessa tehtäville laajennuksille ja uusien ominaisuuksien käyttöönotolle. Ylläpidon ja laajennettavuuden lisäksi suunnitteluvaiheessa tehdään ne oikeat tai väärät ratkaisut, jotka määräävät ohjelman tai ohjelmiston tietoturvan tason. Edellisessä kappaleessa vihjattiin suunnitteluongelmien kulujen kertautuvan ohjelmiston elinkaaren aikana. Tämä vaikuttaa loogiselta esimerkiksi silloin, kun mietitään ketjua jossa huono suunnittelu johtaa vaikeaan laajennettavuuteen joka taas osaltaan luo pohjaa tietoturva- ja muiden ongelmien pesiytymiselle ohjelmistoon mukaan.

Graff ja van Wyk esittävät tietoturvallisen suunnittelun olevan toteuttajan miekka ja kilpi, joilla ohjelmistoa puolustetaan erilaisilta hyökkäyksiltä kohdistuivat ne mistä suunnasta hyvänsä. Puolustuksen lisäksi hyvä suunnittelu on myös vahva perusta, jonka päälle tietoturva voidaan rakentaa.

2.3. Turvallisen suunnittelun kuusi kohtaa

Graff ja van Wyk ovat jakaneet tietoturvallisen suunnittelun kuuteen eri kohtaan. Kohdat on luetteloitu alle ja niitä käydään myöhemmässä tekstissä tarkemmin läpi.

- Arvioi riskit ja uhat
- Tee suunnitelma riskien hallinnan ja realisoitumisen varalle
- Luo mielikuva ohjelman toiminnasta
- Päätä korkean tason toiminnallisuus. Esimerkiksi palvelimen tilattomuus vs. tilallisuus
- Valitse tekniikat, jotka toteuttavat vaatimukset
- Ratkaise tulevaa käyttöä koskevat asiat kuten tietokantojen varmistus ja varmistusten tietoturvallinen suojaus

2.3.1. Riskien ja uhkien arviointi

Luku riskien ja uhkien arvioinnista sisältää yhden kirjan tärkeimmistä neuvoista. Arviointi onnistuu *kysymällä kysymyksiä* joita kirjassa on esitetty suuri joukko. Kysymyksien aihepiiri kattaa tuotetta käyttävän organisaation, suunnittelun alla olevan ohjelmiston itsensä ja ohjelmiston käsittelemän tiedon. Jaon eri aihepiireihin voi epäilemättä tehdä muullakin tavoin. Kysymykset kannattaa esittää niille, joilla on oikeasti jotain menetettävää tai saavutettavaa kyselyn tulosten johdosta sekä oikeaa tietoa vastauksia varten.

Kysymällä kysymyksiä eli tekemällä hyvän selvitystyön suunnittelija voi varautua turvaongelmien ratkaisuun tiedon eikä arvioiden ja arvailujen varalta.

2.3.2. Suunnitelmien teko riskien realisoitumisen ja hallinnan varalle

Riskien hallinta hyppää suoraan käytön aikaiselle hetkelle, ja vaikuttaa ensiksi olevan täysin väärässä kohdassa. Katsottaessa kirjaa kokonaisuutena, samoin kuin kirjoittavat haluavat tarkastella turvallisuutta kokonaisuutena, voidaan huomata kirjan keskittyvän enemmän yrityksen tai organisaation ohjelmistoprosessin tietoturvallisuuden suunnitteluun kuin yksittäisen ohjelmiston ja sen tietoturvan suunnitteluun.

Organisaation suunnitellessa jonkun järjestelmän käyttöönottoa, on järkevää jo alusta lähtien tehdä suunnitelmia riskien hallintaa varten. Jos taas kyseessä on yritys, joka tekee massamarkkinaohjelmistotuotteen, voidaan ajatella ohjelmiston hankkijan olevan

velvollinen päättämään ja asettamaan omat puitteensa ohjelmiston käytön mahdollisesti aiheuttamille riskeille.

2.3.3. Luo mielikuva ohjelmiston toiminnasta

Toinen tärkeistä kirjan antamista neuvoista liittyy mielikuvaan ohjelmiston toiminnasta. On tärkeää, että suunnittelija *ei ajattele* ohjelmistoa liikaa käyttäjän näkökulmasta. Keskittymällä liikaa siihen miten ohjelmistoa käytetään, johtaa siihen, että suunnittelija unohtaa miettiä miten ohjelmistoa vastaan voidaan hyökätä käyttämällä sitä suunnitellusta eroavalla tavalla.

Mielikuvaa ohjelmiston toiminnasta pitää käsitellä monelta eri näkökulmalta, jolloin mahdolliset epäkohdat voidaan parhaiten saada selville. Pelkkä suunnittelu tiettyä maalia ajatellen voi johtaa tietoturvaongelmien syntyyn, koska suunnittelussa ei oteta kokonaisuutta huomioon. Analogiaa löytyy ohjelmistojen testauksessa, jossa testaus suoritetaan vain tiettyihin komponentteihin ilman aikomustakaan pyrkiä täyteen kattavuuteen.

Suunnitellusta poikkeava käyttäytyminen tai siinä pysyminen on myös syynä sille käytännössä tunnetulle ongelmalle, että suunnittelija tai toteuttaja ei yleensä ole paras testaaja valmiille tuotteelle. Ongelmat saadaan esiin parhaiten silloin kun voidaan päästää irti suunnitellusta toimintatavasta, ja ottaa käyttöön hyökkäys tai lähestymistapa jollaista suunnittelija ei ole huomioinut.

Kirjan esimerkki epäonnistuneesta ajatusmallin käytöstä oli TCP:n kolmitiekättely, jossa palvelin varasi aina resursseja yhteydenottoyrityksen aikana. Suunnittelija ei ollut varautunut resurssinvaraushyökkäystä varten. Lopputulos oli sama kuin palvelutiskin jonossa, jossa kiusantekijä varaa itselleen monta jonotusnumeroa ja poistuu paikalta.

2.3.4. Päätä korkean tason toiminnallisuus

Kun oikea kuva ohjelmiston toiminnasta on saatu luotua, voidaan tehdä päätökset korkean tason toiminnasta. Esimerkkinä voidaan pitää palvelinsovelluksen tapauksessa sitä, onko palvelin tilallinen vai tilaton pyyntöjen suhteen. Korkean tason toiminnallisuuteen kuuluvat ohjelmiston sisäinen toiminta, kuten käyttäjien tunnistus ja ohjelmiston vaatimat oikeudet, ohjelmiston toiminta verkossa ja ohjelmiston toiminta hyökkäyksen alla.

Eräs paikallinen esimerkki korkean tason toiminnallisuudesta oli TTY:lle esitelty palvelinhallintaohjelmiston toiminto verkossa. Palvelinta hallittiin verkon kautta, ja palvelimella käskyjä vastaanottavaa komponenttia oli ajettava pääkäyttäjän oikeuksin. Käskyt palvelimella toimivalle hallintakomponentille menivät verkon ylitse salaamattomina.

Korkean tason toiminnallisuus vaikuttaa siltä, että ohjelmisto oli suunniteltu verkkoihin, joihin voitiin (tai ainakin haluttiin asiakkaan voivan) luottaa. Salattu liikenne oli tulossa, eli vaadittu turvallisuus ja ohjelmiston kehityskulut olivat olleet tasapainossa tähän saakka.

Tiedon salaamattomuus voi toimia hallittaessa palvelimia organisaation omassa konesalissa, mutta TTY:n kampusverkko asettaa kovemmat vaatimukset tietoturvalle. Syitä korkealle vaatimustasolle ovat korkeakouluverkolle normaalit asiat kuten verkossa toimivien käyttäjien määrä ja heidän erilaiset kiinnostuksensa. Lisäksi opiskelijoilla on usein tervettä kiinnostusta oman ympäristönsä, tässä tapauksessa verkon, toimintaan.

2.3.5. Valitse tekniikat jotka toteuttavat vaatimukset

Vaatimukset ovat usein ristiriitaisia, ja erittäin usein turvallisuuden taso ja kustannukset ovat suoraan verrannollisia toisiinsa nähden. Se mitä käytännössä halutaan, on tasapaino, jossa turvallisuustaso on riittävä ilman, että niistä aiheutuvat kustannukset nousevat liian suuriksi.

Ennen kun sopiva tasapainotila voidaan löytää, täytyy turvallisuuden tason määrittelemiseksi perehtyä ohjelmiston käyttöympäristöön. Turvallisuuden tasoon vaikuttavat useat seikat, joista yksi tärkeimmistä on turvattavan tiedon kiinnostavuus. Käyttäjien tunnistukseen liittyy aina erehtymisen mahdollisuus, mistä johtuen pitää selvittää se, onko vakavampi ongelma kieltää vai myöntää virheellisesti pääsy johonkin resurssiin.

Myös kallis ratkaisu voi osoittautua turvattomaksi, jos esimerkiksi käyttäjät kokevat turvamenetelmät liian hankaliksi ja normaalia käyttöä rajoittaviksi. Lopputuloksena yleensä huomataan se, että eri arvojen tasapainottaminen muuttuu hankalaksi niiden määrän kasvaessa.

Graff ja van Wyk esittelevät myös lyhyesti SAEM-menetelmän (Security Attribute Evaluation Method), joka pyrkii tuomaan kustannusten lisäksi myös hyödyt selvästi mitattavaksi. SAEM on prosessi, jonka avulla tietoturvahankinnoista päättävät henkilöt voivat selvittää sitä, ovatko hankintojen kustannukset riskejä vastaavia. SAEM ei kuulu suoraan tietoturvallisen ohjelmoinnin piiriin, vaan pyrkii antamaan työkalun päätöksenteon käyttöön. [SAEM]

2.3.6. Tulevaa käyttöä koskevat asiat

Kuudes eli viimeinen kohta käsitteli asioita, jotka liittyvät ohjelmiston turvalliseen käyttöön. Esimerkiksi suunnitteluvaiheessa voidaan ja pitää ottaa kantaa tiedon varmistamiseen ja mikäli mahdollista varmuuskopioiden suojaamiseen. Suunnitteluvaiheessa voidaan miettiä myös tietoturva- ja muiden päivitysten turvallinen lisääminen ohjelmistoon.

Käyttöön liittyviä asioita oli käsitelty kirjan kahdessa seuraavassa luvussa perusteellisemmin.

2.4. Yhteenveto

Turvalliseen suunnitteluun vaadittava pohjatieto vaatii sekä riskien että tulevan käyttöympäristön vaatimusten selvittelyä. Tämän selvitystyön tähdentäminen on luvun parasta antia. Jos suunnittelija ei tunne kohdeympäristön riskejä ja vaatimuksia, joudutaan suunnitteluvaiheessa toimimaan oletusten varassa.

Selvitystyön lisäksi luvun tärkeimpiin asioihin kuuluu muistutus oikeantyyppisen mielikuvan käytöstä suunnitteluvaiheessa. Suunnittelijan pitää olla epäluuloinen ja varautua siihen, että ohjelmistoa vastaan hyökätään juuri siitä suunnasta mistä sitä ei normaalisti käytetä. Jos suunnittelu tapahtuu pelkästään käyttäjän eli normaalin käytön näkökulmasta, hyökkääjälle jää suunnittelussa huomioimatonta liikkumatilaa ja paljon mahdollisuuksia omien tavoitteidensa toteuttamiseen.

Suunnittelijan pitää tuntea sekä kiinnostusta että tervettä epäluuloa ohjelmistonsa tulevaa ympäristöä varten parhaaseen tulokseen päästäkseen.

Lähteet

[GRAFF] Graff, Mark G., ja Kenneth R. Van Wyk.
2003. Secure Coding: Principles & Practices
Sebastopol: O'Reilly & Associates

[SAEM] Security Attribute Evaluation Method: A Cost Benefit Approach International Conference on Software Engineering 2002, (ICSE 2002) Proceedings

3. COPS- protokollakirjasto tietoturvallisen ohjelmoinnin näkökulmasta

Heikki Vatiainen

Tampereen teknillinen yliopisto

3.1. Johdanto

Tässä tekstissä esitellään Tampereen teknillisen yliopiston tietoliikennetekniikan laitoksen IP-palvelunlaatuun liittyneessä projektissa toteutettu COPS-protokollakirjasto tietoturvallisen ohjelmoinnin näkökulmasta.

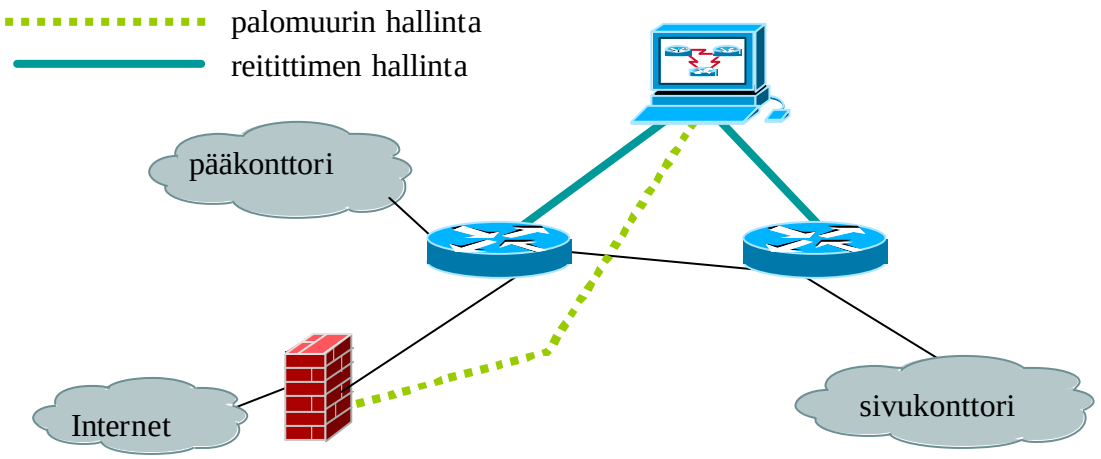
3.2. COPS-protokolla

COPS-määrittelee verkkolaitteen ja hallinta-aseman välisen perusprotokollan verkkolaitteiden konfigurointia varten. COPSia ei ole tarkoitettu verkkolaitteiden konfigurointiin, vaan se luo perustan ylemmän tason protokollille, joita käytetään esimerkiksi reitittimien palvelunlaatuominaisuuksien ja palomuurien sääntöjen konfigurointiin. COPS on määritelty IETF:n RFC:ssä numero 2748, ja sen tarjoamat palvelut COPS-sovelluksille ovat tietoturvan osalta viestien autenttisuuden ja eheyden turvaaminen sekä uudelleenlähetyksen havainnointi. Lisäksi COPS tarjoaa sovelluksilleen tiedon siitä, onko vastapää vielä elossa (keep-alive) [COPS].

Tietoturvallisen ohjelmoinnin kannalta COPSin sijoittuminen protokollapinon alaosiin tekee siitä tietoturvamielessä tärkeän, koska haavoittuvuus useiden sovellusten jakamassa komponentissa voi yhdellä kertaa altistaa useita sovelluksia ja laitteita hyökkäyksille. Hypoteettinen esimerkki verkosta, jossa käytetään COPSia sekä reitittimien että palomuurien konfigurointiin havainnollistaa sitä kuinka koko verkkoinfrastruktuuri voi altistua yhdenkin haavoittuvuuden löytyessä.

Alla olevassa kuvassa on esitelty COPS-protokollaa käyttävää verkkoa. Keskitetyltä hallinta-asemalta konfiguroidaan reitittimien IP-palvelunlaatuun liittyviä asetuksia sekä palomuurin sääntöjä. Molemmat yhteydet käyttävät COPS-protokollaa viestien välittämiseen, mutta sovellukset reitittimissä ja palomuurissa ovat erityyppisiä.

Kuva 9. COPS-protokollaa käyttävä verkko



3.3. COPS-kirjaston toteutus

TTY:llä toteutettu COPS-kirjasto toteuttaa RFC:ssä 2748 määritellyn protokollan. Toteutus on tehty C-kielellä, ja sitä käytettiin tietoliikennetekniikan laitoksella IP-palvelunlaatuun keskittyneen projektin osana tehtyjen demosovellusten yhteydessä.

3.3.1. Toteutuksen päälähtökohdat

COPS-protokolla on tarkoitettu perustaksi monille erityyppisille sovelluksille. COPSin asema yhteisenä komponenttina useille toteutuksille tekee siitä kiinnostavan hyökkäyskohteen, mistä johtuen toteutuksen suunnitteluvaiheessa tietoturvalliseen ohjelmointiin kiinnitettiin alusta lähtien huomiota. Seminaarin ensimmäisessä esityksessä käsiteltiin tietoturvaongelmien korjausten kustannuksia ohjelmiston elinkaaren aikana, ja kuten yleisesti on tiedossa, ongelmien korjaaminen jo suunnitteluvaiheessa on kaikista halvinta. Toteutusvaiheessa COPS-kirjaston käytön laajuuttaa ei osattu vielä arvioida, mutta koska kyseessä oli uusi toteutus, tietoturvallisuuden suunnittelu kirjastoon mukaan oli käytännössä yhtä halpaa ja helppoa kuin sen pois jättäminenkin.

Tietoturvallisen ohjelmoinnin osalta kirjaston toteutus voidaan jakaa kahteen osaan, jotka ovat kirjaston käyttäjälle näkyvä osa sekä kirjaston sisäinen toteutus. Kirjaston käyttäjä huomaa kirjaston toteuttavan olio-ohjelmointikielistä tuttua tiedon piilotusta sekä omana ominaisuutenaan vaatimuksen sille, missä järjestyksessä kirjaston tarjoamia funktioita voidaan kutsua. Kirjaston sisäisessä toteutuksessa turvallisuuden kannalta on keskitetty dynaamisen muistinvarauksen johdonmukaiseen käyttöön, verkosta luettavan datan epäluuloiseen ja varovaiseen käsittelyyn sekä kirjaston tekemiseen turvalliseksi myös siinä tapauksessa, että sitä käytetään sovelluksissa jotka käyttävät säikeitä (threads).

3.3.2. Turvallisuus COPS-kirjaston käyttäjän näkökulmasta

COPS-kirjasto on suunniteltu olemaan epäluuloinen verkon lisäksi myös käyttäjäänsä kohtaan. Toisaalta kirjasto ei yritä tehdä epäluuloisuudesta käyttäjää ärsyttävää, jotta käyttäjä ei tuntisi tarvetta kiertää kirjaston tarjoamia rajapintoja. Suunnitteluvaiheessa verkkoprotokollille haettiin muutamia yleisiä toimintoja, joista voidaan koota esimerkiksi alla näkyvä lista tilanteelle, jossa verkosta halutaan lukea protokollasanoma:

- Luetaan verkosta COPS-sanoma.
- Tutkitaan sen tyyppi. Usein tämä ja ensimmäinen askel liittyvät toisiinsa.
- Tarkistetaan onko sanoma tyyppiä vastaavan määrittelyn mukainen.
- Poimitaan sanomasta eri kentät joiden määrä tavallisesti riippuu tyypistä.
- Vapautetaan sanoman käyttämä muisti.

Kirjaston epäluuloisuus ja käytön miellyttävyys sovitettiin toisiinsa suunnittelemalla kirjastolle rajapinta, joka toteuttaa yllä olevassa listassa mainitut asiat. Koska COPS-sanomien käsittely kaikille COPSia käyttäville sovelluksille on aina sama, jokaiseen sovellukseen ei tarvitse toteuttaa samoja askeleita, vaan ne jätetään COPS-kirjastolle. Tällä pyritään tekemään COPS-kirjastosta käyttäjälle miellyttävä ja hyödyllinen. Samal-

la saadaan toteutettua epäluuloisuus käyttäjää kohtaan tekemällä COPS-sanomasta abstrakti tietotyyppi, jonka sisältö on käyttäjälle tuntematon, ja joka yhdessä kirjaston toteutuksen kanssa pakottaa käyttäjää kutsumaan kirjaston funktioita yllä mainitun listan antamassa järjestyksessä.

Edellä mainittu abstrakti tietotyyppi ja siihen liittyvä tiedon piilottaminen mahdollistavat sen, että käyttäjä ei missään vaiheessa esimerkiksi pääse suoraan verkosta luettuun dataan käsiksi, vaan sanomien kentät pitää poimia erillisellä funktiolla. Tällä funktiolla on mahdollisuus varmistaa se, että esimerkiksi pituuskentät eivät osoita paketin rajojen ulkopuolelle. Huolellinen käyttäjä voisi tehdä tarkistuksen itsekin, mutta nyt sekä huolellinen että huoleton käyttäjä voivat olla välittämättä näistä asioista.

3.3.3. Esimerkki COPS-kirjaston käytöstä

Kuvassa 2 on esitelty typistetysti COPS-kirjaston käyttäjän toiminta luettaessa protokollasanoma verkosta.

```
static int handle_packet(cops_conn *pdp_conn)
{
    cops_msg *msg;
    int ret;
    struct cops_common_hdr hdr;
    long acctimer;

    msg = cops_alloc_msg();
    if (msg == NULL) return -COPS_ERR_NOMEM;

    ret = cops_read_msg(msg, pdp_conn);
    ret = cops_check_incoming_msg(msg);
    ret = cops_get_common_hdr(msg, &hdr);
    switch (hdr.op_code) {
    case COPS_MSG_KA:
        cops_handle_ka(msg);  /* The COPS lib takes care
of keep-alives */
        break;
case COPS_MSG_DEC:

 ret = handle-dec(msg);
```

Kuva 10. . COPS-protokollasanoman vastaanotto

COPS-protokollasanoman abstrakti tietotyyppi on cops_msg, jonka käyttäjä joutuu varaamaan COPS-kirjastolta. Se mitä cops_msg sisältää, on tiedossa ainoastaan kirjaston sisällä eikä sen määritelmää löydy kirjaston rajapinnan määrittelevistä otsikkotiedostoista. Tästä johtuen kirjaston käyttäjä voi ainoastaan käyttää cops_msg:tä kahvana, joka annetaan kirjaston funktioille. Koska käyttäjä ei tunne cops_msg:n rakennetta, hän ei voi päästä sen kautta käsiksi mihinkään verkosta luettuun dataan tai cops_msg:n sisäiseen tilaan.

Cops_msg:n sisäinen tilatieto pakottaa esimerkiksi sen, että kutsuttuaan funktiota cops_read_msg, käyttäjä ei seuraavana voi kutsua funktiota cops_get_common_hdr päästäkseen viestin tyyppiin käsiksi. Cops_msg toteuttaa yksinkertaisen tilakoneen, joka

huolehtii kirjaston toteuttajan suunnittelemien funktioiden kutsumisen ennaltamäärätyssä järjestyksessä. Edellisessä esimerkissä funktiota cops_get_common_hdr voidaan kutsua ainoastaan silloin, jos funktiota cops_check_incoming_msg on jo kutsuttu. Mikäli käyttäjä jättää paluukoodit tutkimatta tai ei välitä niistä, tilasiirtymää ei tapahdu kutsuttaessa vääriä funktioita.

Kuvassa 3 ei esitetä sanoman vapauttamista, joka tehdään kutsumalla funktiota cops_free_msg. Typistetyssä esityksessä ei myöskään näy paluukoodien tarkastaminen, vaan ret-muuttujan arvo jätetään huomioimatta.

Esimerkissä on myös käytetty tietotyyppiä cops_common_hdr, jonka instanssi hdr sisältää tiedon COPS-viestin tyypistä. Vaikka tiedot ovat samat kuin mitä verkosta luetussa viestissä on, ne ovat ainoastaan kopio alkuperäisen viestin kentistä. Käyttäjä ei tälläkään tavalla pääse muuttamaan alkuperäistä viestiä.

Esimerkissä näkyy myös COPS-kirjaston toteuttama Keep-Alive-viestien käsittely. Kun verkosta saatu sanoma on tunnistettu Keep-Alive-tyyppiseksi, se voidaan antaa suoraan COPS-kirjaston käsiteltäväksi.

3.3.4. Turvallisuus COPS-kirjaston sisäisessä toteutuksessa

COPS-kirjaston sisäinen toteutus turvallisuuden kannalta tarkoittaa tässä tekstissä yksinkertaisesti niitä hyviä toimintatapoja, joita kirjaston toteuttavan koodin kirjoittamisessa on käytetty. Tekstissä mainittiin aiemmin erityistä huomiota kiinnitetyn dynaamiseen muistinhallintaan, verkosta luettavan datan käsittelyyn ja varautumista kirjaston käyttöön osana säikeitä käyttävää sovellusta. Näitä kolmea osa-aluetta on käsitelty alla omina kokonaisuuksinaan.

3.3.4.1. Dynaamisesti varatun muistin käsittely

COPS-kirjasto antaa käyttäjälle kaksi erityyppistä tietotyyppiä, joista ensimmäinen kuvaa yhteyttä naapuriolioon, esimerkiksi hallinta-aseman yhteyttä kontrolloitavaan palomuuriin, ja toinen kuvaa parhaillaan käsiteltävää COPS-sanomaa. Molempiin liittyvää dynaamista muistia hallitsee COPS-kirjasto, jolta käyttäjä pyytää tietotyypit, ja jolle käyttäjä palauttaa tietotyypit käytön jälkeen.

Esimerkkinä voidaan käsitellä lähetettävän viestin rakentamista, jossa käyttäjä lisää cops_msg-tyyppiseen viestiin omaa dataansa. Kirjasto ei voi tietää datan määrää ennalta käsin, joten viestin rakentamisesta huolehtiva funktio huolehtii viestin koon dynaamisesta kasvattamisesta. Kun viesti on lähetetty, käyttäjän ei tarvitse huolehtia varatun muistin määrästä, vaan cops_msg pitää kirjaa varatuista muistialueista, ja funktio cops_free_msg osaa tämän tiedon avulla vapauttaa kaiken dynaamisesti varatun muistin.

3.3.4.2.Verkosta luettavan datan käsittely

Yksi seminaarin ensimmäisessä osassa esitellystä tietoturvallisen suunnittelun viidestä kohdasta oli oikean ajatusmallin muodostaminen. Suunnittelija on pystyttävä ajattelemaan asioita hyökkääjän kannalta eikä käyttäjän kannalta. Käyttäjän kannalta ajateltuna verkosta luettavan datan käsittely on äärimmäisen helppoa: luetaan vastaanotetusta viestistä protokollamäärittelyn mukaiset arvot!

Ensimmäinen virhe tapahtuu jo siinä vaiheessa kun oletetaan, että verkosta todellakin tulee viesti eikä esimerkiksi satunnaista tai tahallista roskaa. Mikäli verkosta todellakin tulee viesti, saattaa siinä olla esimerkiksi tahallisesti virheellisiä pituus- tai viestityyppien kenttiä.

COPS-kirjaston toteutus pyrkii tekemään vastaanotetulle sanomalle mahdollisimman kattavan tarkastelun. Mahdollisia tarkastettavia asioita ovat esimerkiksi seuraavat: viestin tyypin pitää olla tunnettu, pituuskentässä ei voi olla ylisuuri luku ja viestin sisältämien vaihtelevanpituisten kenttien pituudet eivät voi ulottua viestin lopun yli.

COPS-kirjasto voi tutkia viestiä tietylle tasolle saakka, mutta esimerkiksi kahden palomuuriolion keskustellessa keskenään kaikki niiden vaihtama hyötykuorma on COPS-kirjastolle läpinäkymätöntä. COPS-kirjastoa käyttävät sovellukset voivat kuitenkin varmistua siitä, että kaikki niiden saamat viestit ovat COPS-protokollan puolesta virheettömiä.

COPS-kirjasto pyrkii myöskin turvaamaan omalta osaltaan verkosta luettavan datan oikeellisuutta tekemällä tarkastuksia käyttäjän lähettämälle datalle. Koska käyttäjä ei voi olla varma, että vastapuoli on tehnyt näitä tarkastuksia, niistä saatava hyöty liittyy lähinnä ohjelmointivirheiden havainnointiin.

3.3.4.3.COPS-kirjaston tuki säikeille

COPS-kirjaston tuki säikeille oli helppo suunnitella ja toteuttaa: kirjasto käyttää ainoastaan dynaamisesti varattua muistia mutta ei käytä globaaleja muuttujia. Rajoituksena on ainoastaan se, että mahdolliset säikeet eivät jaa COPS-kirjastolta dynaamisesti varaamiaan kahvoja.

Käyttäjän kannalta tämä tarkoittaa sitä, että COPS-kirjaston funktiot ovat re-entrantteja, eli esimerkiksi kaksi säiettä pystyy samanaikaisesti kutsumaan cops_alloc_msg-funktiota varatakseen kahvan lähetettävälle tai vastaanotettavalle viestille. Käytännön ongelmat ei-re-entreanttien funktioiden kanssa näkyvät lähinnä ohjelmiston sekoamisena tai kaatumisena satunnaisina hetkinä varsinkin silloin kun tapahtumia on paljon.

3.4. Onnistumiset ja epäonnistumiset

COPS-kirjastoa käytettiin tietoliikennetekniikan laitoksen projektissa IP-palvelunlaatusovellusten toteuttamiseen. COPSia käyttävät sovellukset hyötyivät COPS-kirjaston rajapinnan vaatimuksista lähinnä sovellusten omien ohjelmointivirheiden totetamisessa. Tietoturvan kannalta kirjaston ominaisuudet eivät päässeet oikeuksiinsa, sillä laboratorioympäristössä ne eivät koskaan altistuneet oikealle käytölle, jossa turvallisuusnäkökohdat ovat tärkeitä. Tärkeää oli kuitenkin se, että tietoturvaominaisuudet eivät osoittautuneet haitallisiksi eivätkä ärsyttäviksi käyttää.

Suurin hyöty tekijälle oli tietoturvallisen ohjelmoinnin mallin kokeilu ja siitä saadut hyvät kokemukset, jotka antavat pohjaa mahdollisille myöhemmille toteutuksille.

Epäonnistumisiksi voidaan katsoa COPS-protokollan tietoturvaominaisuuksien toteuttaminen vasta muun toteutuksen jälkeen. COPS sisältää mahdollisuuden liittää jokaiseen viestiin COPS-kerrokselle kuuluvaa lisätietoa, jolla voidaan varmistaa se, että viesti tulee oikealta lähettäjältä, viestin sisältöä ei ole muutettu matkalla ja se, että viestiä ei yritetä lähettää uudelleen (replay attack).

Näitä ominaisuuksia ei suunniteltu toteutukseen mukaan heti alussa, ja niiden lisääminen valmiiseen toteutukseen oli hankalampaa kuin niiden mukaan ottaminen heti toteutuksen alussa.

3.5. Yhteenveto

Tarkastelun alla ollut COPS-kirjaston toteutus sisältää useita seminaarissa käsiteltyjä tietoturvalliseen ohjelmointiin liittyviä asioita. Suurimmaksi tietoturvaongelmaksi nähtiin verkosta saapuvien viestien käsittely, joten kirjasto pakottaa käyttäjän käsittelemään verkosta saapuvat paketit tietyllä turvallisella tavalla. Tavoitteena oli se, että vahingossa tai tahallaan korruptoituneet COPS-viestit voidaan tunnistaa ja käsitellä hallitusti.

Kirjaston käyttöympäristöksi suunniteltiin heti alusta lähtien globaali Internet, jossa COPS-toteutus on mahdollisimman suuren uhan alla. Suunnittelemalla toteutus suljettua verkkoa varten kirjasto olisi ehkä voitu saada pienemmäksi, mutta näin saadut hyödyt olisivat olleet lähinnä marginaalisia. Lisähyötyä viestien tarkalla tarkastamisella saatiin siitä, että osa kirjastoa käyttävien sovellusten ongelmista saatiin paljastettua tällä tavoin.

Lähteet

[COPS] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan and A. Sastry, "The COPS Protocol", IETF RFC: 2748, January 2000.

4. Turva-arkkitehtuuri

2.2.2004, versio 0.4

Karri Huhtanen

<karrih@cs.tut.fi>

Tampereen teknillinen yliopisto

Tietoliikennetekniikan laitos

4.1. Turva-arkkitehtuurin määritelmä

Turva-arkkitehtuuri on määritellyn turvatarpeen täyttämiseen tähtäävä prosessi, jossa valikoimalla suunnitteluelemettejä ja -periaatteita pyritään luomaan runko korkean tason suunnitteluperiaatteita ja -päätöksiä. Tämä runko auttaa sitten ohjaamaan myöhempiä järjestelmän tai ohjelmiston kehityksen vaiheita kuten suunnittelua tarjoamalla suunnittelijoille varmat vastaukset perustavanlaatuisiin kysymyksiin, kuten miksi ja mikä on ollut tavoite, jota on tätä tietoturvapäätöstä tehdessä haettu. Suunnittelijat ja toteuttajat pystyvät näin hyvän tietoturva-arkkitehtuurin pohjalta näkemään, mikä on ollut järjestelmän tai ohjelmiston ideana ja pystyvät näin myös sopeuttamaan omat päätöksensä tämän ohjenuoran mukaisiksi.

4.1.1. Mitä turva-arkkitehtuuri ei ole

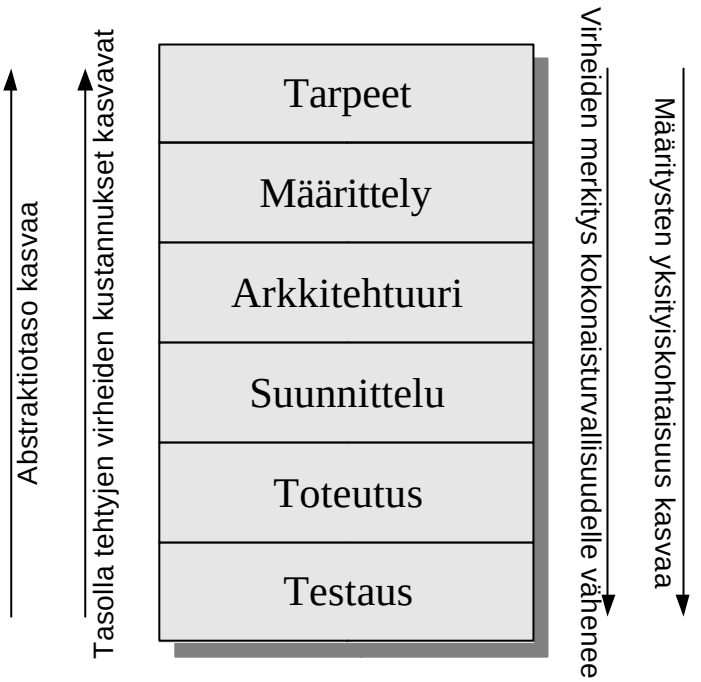
Turva-arkkitehtuuri ei ole asia, joka kannattaa sitoa tiukasti tiettyyn osaan järjestelmien ja ohjelmistojen suunnittelua. Hyvä turva-arkkitehtuuri ottaa huomioon yksittäisen järjestelmän tai ohjelmiston arkkitehtuurin lisäksi myös ympäröivät elementit ja soveltuu näin käytettäväksi tehtävien suunnittelu- ja toteutusratkaisujen pohjana niin verkko-, järjestelmä- ja ohjelmistosuunnittelunkin eri tasoilla. Turva-arkkitehtuurin määrittteleminen vain tietyn osa-alueen pohjalta voi johtaa sen sovellettavuuden ja jopa turvallisuuden heikkenemiseen, kun ohjelmistojen ja järjestelmien väliset liitokset ja suhteet saattavat jäädä huomioimatta. Turva-arkkitehtuuriin eivät kuulu kuitenkaan turvapolitiikan, -standardien ja -käytäntöjen määrittäminen vaan nämä määritykset ja päätökset pitäisi tehdä erillään ja ylemmällä tasolla. Kun ne on tehty, niitä voidaan käyttää turva-arkkitehtuurin suunnittelun ohjaamiseen.

4.1.2. Turva-arkkitehtuurin hyödyt

Hyvän turva-arkkitehtuurin uudelleenkäytettävyys ja sovellettavuus tulikin jo tekstistä ilmi. Hyötynäkökulmasta voidaan ajatella, että kun pätevä, uudelleenkäytettävä turva-arkkitehtuuri on saatu luotua, ei tarvitse jokaista projektia, järjestelmää tai ohjelmistoa varten aloittaa rakentamaan kaikkea tyhjästä uudestaan. Saman pätevän turva-arkkitehtuurin soveltaminen myös takaa parhaimmillaan sen, että esimerkiksi elementtejä yhdistellessä kokonaisuuden turvallisuus pystytään takaamaan sen sijaan, että se pitäisi aina erikseen arvioida. Yhtenäisen, uudelleenkäytetyn turva-arkkitehtuurin hyötyjä ajatellessa on kuitenkin myös huomattava kääntöpuoli eli se, että onko turva-

arkkitehtuuri todellakin pätevä. Vaillinainen turva-arkkitehtuuri voisi riippuen uudelleenkäytön asteesta vaarantaa esimerkiksi kokonaisen ohjelmistoperheen ja siitä riippuvat järjestelmät kerralla, jos virhe löytyisi turva-arkkitehtuurista.

Ehkä vielä selvempi hyöty turva-arkkitehtuurista on liiketaloudellinen hyötynäkökulma, joka on täsmälleen sama kuin yleensäkin ohjelmistotuotannossa tai järjestelmäintegraatiossa – miksi kannattaa tehdä asiat kunnolla alusta lähtien? Siksi, että myöhemmin niiden korjaaminen tai tekeminen on huomattavasti kalliimpaa. Tilannetta havainnollistaa seuraava kuva, joka perustuu ohjelmistoprojektin eri vaiheissa löydettyjen ja tehtyjen virheiden kustannuksille.



Kuva 11. Ohjelmistoprojektin eri vaiheissa löydettyjen ja tehtyjen virheiden kustannukset

Kuten kuvassa esitetään, kun abstraktiotaso nousee, nousee myös kyseisillä tasoilla tehtyjen virheiden korjaamisen kustannukset. Mitä korkeammalla tasolla virhe on, sitä useampi alempi taso on käytävä läpi virheen korjaamiseksi ja tämä lisää projektin kustannuksia. Esimerkiksi satunnaisen puskuriylivuodon korjaaminen ja testaaminen saattaa vaatia vain rajatun määrän ohjelmointia ja testausta, kun taas virhe ohjelmiston turva-arkkitehtuurissa voi vaatia koko ohjelmiston läpikäymisen ja ehkä jopa täyden uudelleentoteutuksen, koska alemman tason suunnittelu- ja toteutusratkaisut on tehty toteuttamaan ylemmällä tasolla määriteltyjä asioita. Mitä ylemmällä tasolla virhe on siis tehty, sen vaarallisempi se on järjestelmän kokonaisturvallisuudelle.

Turva-arkkitehtuurilla, kuten yleensäkin suunnittelulla ennen tekemistä, voidaan välttää tällaisia turhia kustannuksia ja resurssien hukkaamista, joka muuten on niin ominaista ainakin aloittelevalle ohjelmistotuotannolle. Jos turva-arkkitehtuuri on suunniteltu riittävän hyvin, se pystyy myös auttamaan tilanteessa, jossa alemmalla abstraktiotasolla on tehty virheitä huomioimalla ja määrittelemällä esimerkiksi tavan reagoida löydettyihin virheisiin toteutuksessa, suunnittelussa tai testauksessa. Tämä voidaan tehdä esimerkiksi yksinkertaisesti huomioimalla, että mistään osasta ohjelmistoa ja järjestelmää ei saada

koskaan täydellisesti turvallista, mutta turva-arkkitehtuurin avulla virhekään jossain osassa ei aiheuta kokonaisuuden vaarantumista tai ainakin se voidaan korjata nopeasti hajottamatta kokonaisuutta.

4.1.3. Turva-arkkitehtuurin määrittelyn periaatteet

Lähteenä ollut Graffin ja Van Wykin Secure coding, principles and practices -kirja [GraWyk03] muotoili turva-arkkitehtuurin määrittelyn periaatteet karkeasti suomentaen seuraavasti:

1. Turva-arkkitehtuurin määrittelyn periaatteet
2. Aloita kysymysten teolla
3. Määrittele tavoite ennen kiirehtimistä
4. Päätä kuinka turvallinen on riittävän turvallinen
5. Käytä vakiintuneita tekniikoita
6. Tarkasta käyttämäsi oletukset
7. Ota turvallisuus huomioon jo alusta
8. Pidä vihollinen mielessä suunnitellessasi
9. Tunnista ja kunnioita luottamusketjuja
10. Muista vähimpien oikeuksien periaate
11. Tarkista käytäntö ennen toimintojen suorittamista
12. Huolehdi riittävästä viansietoisuudesta
13. Käsittele virhetilanteita huolellisesti
14. Lahoa kauniisti
15. Kaadu turvallisesti
16. Valitse turvalliset oletusarvot ja –toiminnat
17. Yksinkertainen on kaunista
18. Paloittele perusteellisesti
19. Piilossa et ole turvassa
20. Minimoi tilojen määrä
21. Älä kiusaa käyttäjiä liikaa
22. Varmista vastuullisuus

23. Tee resurssien kulutukselle rajat

24. Tapahtumien on oltava rekonstruktoitavissa

25. Eliminoi heikoin lenkki

26. Rakenna kerrospuolustus

27. Turvaa kokonaisuus

28. Kaikkea koodia ei tarvitse eikä pidä tehdä itse

29. Vältä hyllyohjelmistoja

30. Älä polje demokratiaa

31. Mikä unohtui

Useimmat periaatteista ovat jo ohjelmistotuotantoa opiskelleille tai sen parissa työskenneille jo vanhoja tuttuja ja eivätpä ne paljoa poikkeakaan maalaisjärjellä ajatelluista tai perinteisistä ohjelmistosuunnittelun periaatteista. Kirjan tekijät pitivätkin listaa lähinnä muistin apuna niistä kaikista asioista, joita turva-arkkitehtuurin ja yleensäkin ohjelmistojen ja järjestelmien suunnittelussa pitäisi pitää mielessä. Viimeinen periaatekin on puettu näissä suomennetuissa versioissa kysymyksen muotoon: Mikä unohtui? Tämä on aika osuva kysymys, sillä useasti jotkut mainituista periaatteista unohdetaan suunnitteluvaiheessa joko vahingossa tai sitten jopa tahallaan resurssipulan, politiikan tai kustannusten takia. Tästä voidaan kuitenkin kärsiä jatkossa, kuten kuva antoi ymmärtää. n verkkojen pääsynvalvontajärjestelmän arkkitehtuuri

5. Esimerkki: Tampereen teknillisen yliopiston julkisten alueiden verkkojen pääsynvalvontajärjestelmän arkkitehtuuri

Karri Huhtanen

<karrih@cs.tut.fi>

Tampereen teknillinen yliopisto

Tietoliikennetekniikan laitos

5.1. Tausta

Langattomien verkkojen lisääntynyt ja helpottunut käyttöönotto johti Tampereen teknillisellä yliopistolla (TTY) tilanteeseen, jossa eri laitokset, ryhmät ja jopa yksittäiset henkilöt olivat ottaneet käyttöönsä langattomia tukiasemia ja yhdistäneet niitä vaihtelevin käytännöin ja asetuksin TTY:n verkkoon. Osassa oli käytössä pääsynvalvonta perustuen esimerkiksi MAC-osoitelistoihin – osa verkoista oli täysin avoinna mahdollisille tunkeutujille. Tätä ongelmaa ratkaisemaan perustettiin TTY:n tietohallinnon ja Tietoliikennetekniikan laitoksen välinen yhteistyöprojekti [TutPubAc], jonka arkkitehtuurisuunnitelijana kirjoittaja toimi.

Kyseisessä projektissa ei käytetty hyödyksi Graffin ja Van Wykin muistilistaa (luku 3.1.3) vaan arkkitehtuurisuunnittelijan ohjelmistotuotannon ja tietoliikennetekniikan peruskoulutusta sekä käytännön kokemusta laitevalmistajan ja operaattorin palveluksessa toimimisesta. Tehdyissä ratkaisuissa ja tehdyissä virheissä voidaan kuitenkin havaita useimpien Graffin ja Van Wykin muistilistan sääntöjen pätevyys käytännön elämässä sekä se, kuinka turva-arkkitehtuuri on oikeastaan osa järjestelmän arkkitehtuuria.

5.2. Tarpeiden määrittely

Van Wykin ja Graffin muistilistan ensimmäiset kaksi ohjetta ovat:

- Aloita kysymysten teolla
- Määrittele tavoite ennen kiireht imistä

Aloitettaessa projektia haastattelin tietohallinnon johtoa sekä teknisiä henkilöitä, esittelin erilaisia vaihtoehtoja arkkitehtuuriksi ja toiminnallisuudeksi ja kyselin millaisia vaatimuksia ja tavoitteita arkkitehtuurille oli asetettava. Tuloksena sain määriteltyä kyseisen arkkitehtuurin neljä pääsytapaa verkkoon:

- Opiskelijan pääsy verkkoon
- Henkilökunnan pääsy verkkoon
- Vierailijan pääsy verkkoon
- Verkkovierailijan pääsy verkkoon

Näiden neljällä pääsytavalla oli jokaisella omat turvallisuusvaatimuksensa ja –tarpeensa, mutta joilla pystyttiin palvelemaan suurinta osaa langattoman verkon käyttäjistä.

5.3. Tarpeista tavoitteisiin

Kun kysymykset oli tehty ja tarpeet olivat selvillä, pystyttiin etenemään koko järjestelmän arkkitehtuurin tavoitteiden (*Graff Van Wyk 2: Määrittele tavoite ennen kiirehtimistä*) määrittelyyn, joita olivat:

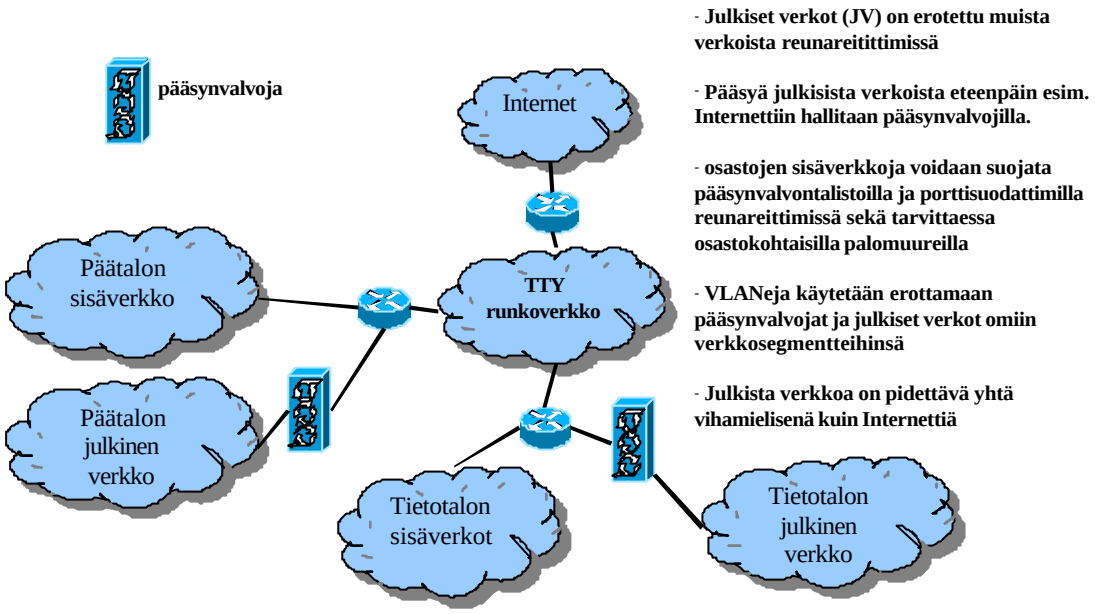
- Riittävä turvallisuus (Graff, Van Wyk 3: Päätä kuinka turvallinen on riittävän turvallinen)
    - Henkilökunnalle vahvasti autentikoitu salattu pääsy osaston verkkoon
    - Opiskelijoille perusautentikointi ja rajatut palvelut
    - Vierailijoille ja verkkovierailijoille mahdollisuus käyttää VPN-ohjelmistoja kotiverkkoonsa suuntautuvien yhteyksien suojaamiseen
- Joustavuus, päivitettävyys, skaalautuvuus
    - Arkkitehtuurin pitää mahdollista uusien palveluiden, verkkoelementtien ja päivitysten joustava lisäys verkkoon
    - Arkkitehtuurin ei pidä rajoittaa verkon kasvua ja skaalautuvuutta
- Yhteensopivuus, avoimuus, standardit (Graff, Van Wyk 4: Käytä vakiintuneita tekniikoita)
    - Arkkitehtuurin täytyy tukea sekä kaupallisia että ei-kaupallisia verkkoelementtejä standardirajapintojen kautta
    - Avoimia standardeja ja rajapintoja suositaan
    - Suljettuja, laitteistovalmistajakohtaisia standardeja pitää välttää
- Käytettävyys
    - Peruspääsyn ei pidä vaatia erityisiä asiakasohjelmistoja, -laitteistoja tai käyttöjärjestelmää käyttäjän päätelaitteelta (*Graff, Van Wyk 16: Yksinkertaisuus on kaunista, Graff, Van Wyk 20: Älä kiusaa käyttäjää liikaa)*

Pari sääntöä kaipaa kirjoittajan mielestä lisätarkennusta tässä yhteydessä: Vakiintuneiden tekniikoiden käyttö tässä arkkitehtuurissa tarkoitti mahdollisimman tunnettujen ja yhteensopivien hyväksi todettujen tekniikoiden käyttöä. Avoimien ja standardien tekniikoiden katsottiin tässä tapauksessa täyttävän vaatimukset paremmin kuin suljettujen johtuen koko järjestelmän toimialueen laitteisto- ja käyttäjäkannan sekalaisuudesta.

Käyttäjien kiusaamisen välttäminen on myös tärkeää turva-arkkitehtuuria tai mitä tahansa arkkitehtuuria suunniteltaessa. Jos turva-arkkitehtuurin pohjalta toteutettu järjes-

telmä tuntuu liian monimutkaiselta ja hankalalta käyttää, pyrkivät käyttäjät valikoimaan helpomman tavan, joka voi tarkoittaa samalla myös tietoturvallisuuden kannalta huonompaa tapaa tai sitten tässä tapauksessa pääsynvalvontajärjestelmän kiertämisen yrittämistä.

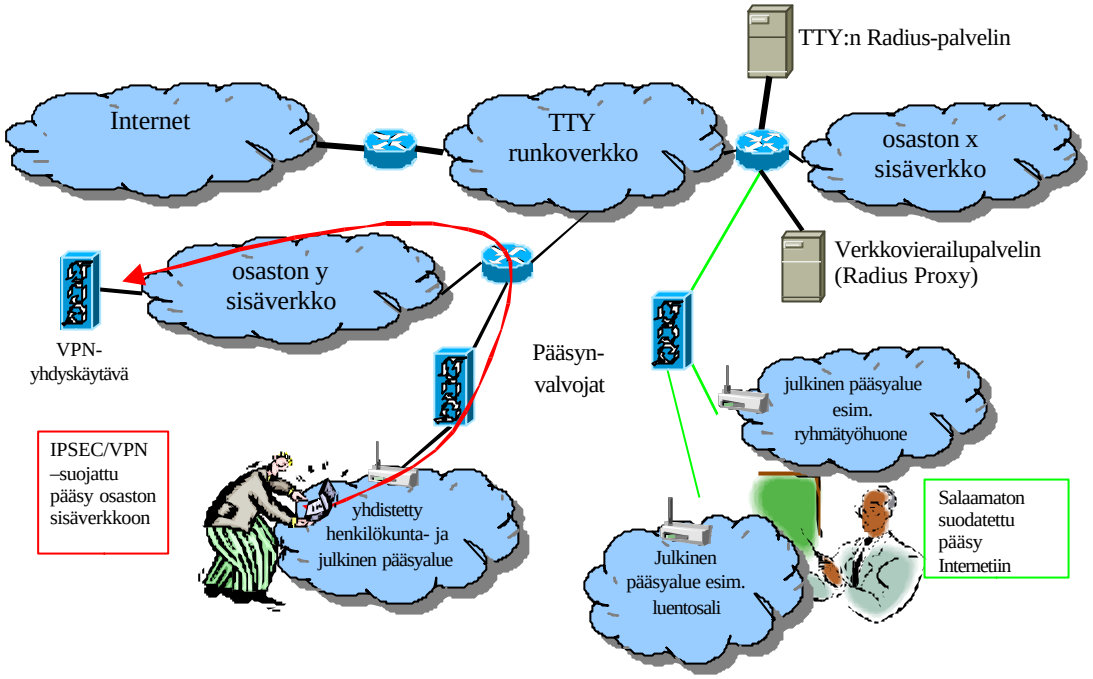
5.4. Tavoitteista verkon arkkitehtuuriin



Kuva 12. Verkon yleiskuva

Tavoitteiden määrittelyn jälkeen pystyttiin siirtymään järjestelmän verkkoarkkitehtuurin määrittelyyn. Verkkoarkkitehtuurissa käytettiin vakiintunutta suunnitteluratkaisua (*Graff, Van Wyk 4: käytä vakiintuneita tekniikoita*), jossa julkiset verkot erotettiin palomuurein, reititinsuodatuksin ja pääsynvalvojin varsinaisista sisäverkoista ja Internetistä (*Graff, Van Wyk 17: Paloittele perusteellisesi*). Tällä tavalla pystyttiin muodostamaan tavallaan kerrospuolustus (*Graff, Van Wyk 25: Rakenna kerrospuolustus*), jossa pääsynvalvojan murtuessa muut verkkoelementit vielä suojaavat muita verkkoja tunkeutumiselta. Pääsynvalvoja valittiin näin järjestelmän oletetuksi murtumakohdaksi (Graff, Van Wyk 13: Lahoa kauniisti), jolloin muut järjestelmät saatiin suunniteltua ottaen huomioon tämä arkkitehtuurisuunnittelun kohta.

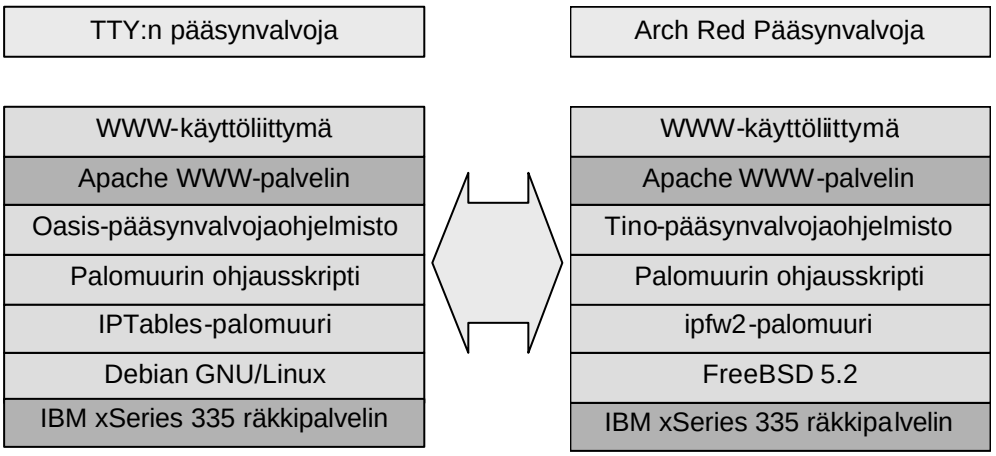
5.5. Verkon arkkitehtuurista verkkoelementtien arkkitehtuuriin



Kuva 13. Verkkoelementtien arkkitehtuurit

Verkkoelementtejä, joita projektin aikana toteutettiin alusta alkaen oli vain yksi, pääsynvalvoja. Muut verkkoelementit (VPN-yhdyskäytävä, TTY:n RADIUS-palvelin, Verkkovierailupalvelin) olivat joko ennestään olemassa tai toteutettiin vain integroimalla (*Graff, Van Wyk 27: Kaikkea koodia ei tarvitse eikä pidä tehdä itse*) erilaisia open-source-ohjelmistoja ja –käyttöjärjestelmiä (*Graff, Van Wyk 30, Vältä hyllyohjelmistoja*).

5.6. Pääsynvalvojan arkkitehtuuri



Kuva 14. Pääsynvalvonnan arkkitehtuuri

Pääsynvalvojaksi arvioitiin muutamia eri vaihtoehtoja. Valmiita tuotteita on esimerkiksi saatavilla Nokialta, Nomadixilta ja Vernier Networksilta. Nämä kaikki kuitenkin olivat suljettuja ja kalliitakin ratkaisuja. Joustavuuden ja avoimuuden vuoksi tein päätöksen

suunnitella ja toteuttaa TTY:n pääsynvalvojan itse integroimalla sen kuvassakin näkyvistä opensource-ohjelmistoista. Ohjelmistokomponentit valitsin niiden toteuttamien rajapintojen selkeyden ja avoimuuden perusteella. Tämän vuoksi ne olikin helppo integroida kuvan mukaiseksi kerrosrakenteeksi niin, että jokaisen komponentin välissä oli selkeä rajapinta (*Graff, Van Wyk 17: Paloittele perusteellisesti*). Laitteistoalusta puolestaan tuli valittua TTY:n tietohallinnon ehdotuksen perusteella.

Tuloksena verkkoelementin arkkitehtuurisuunnittelusta syntyi ensimmäinen versio pääsynvalvojasta (TTY:n pääsynvalvoja), joka otettiin pilottikäyttöön ja samalla testaukseen suoraan TTY:n kirjaston langattoman verkon pääsynvalvonnassa. Tässä vaiheessa erään ohjelmistokomponentin virheitä ei vielä huomattu tai pystytty huomaamaan.

5.7. Pääsynvalvontaohjelmiston arkkitehtuuri

TTY:n pääsynvalvoja perustui Oasis-nimiseen pääsynvalvojaohjelmistoon, joka oli käytännössä palvelinohjelmisto, joka otti vastaan autentikointitietoja WWW-käyttöliittymältä ja varmistaen niiden oikeellisuuden käski palomuuriskriptiä avaamaan palomuurisäännöt tietylle päätelaitteelle. Tämän jälkeen ohjelmisto jäi seuraamaan kyseisen päätelaitteen olemassaoloa ARP-viestien avulla. Tämän ohjelmisto teki säikeiden ja useampien erillisten kirjastojen avustuksella.

Vaikka Oasiksessa näin olikin toteutettu Graffin ja Van Wykin sääntöä 27: Kaikkea koodia ei tarvitse eikä pidä tehdä itse, siinä oli kuitenkin unohdettu tarkistaa koko kokonaisuuden turvallinen toiminta. Tuloksena oli, että Oasis käytti säikeitä ja ei-säieturvallisia kirjastoja toiminnallisuutensa toteuttamiseen. Tämä ja pari muuta virhettä ohjelmiston koodissa aiheuttivat sen, että ajoittain riittävän suurella autentikoitumisten määrällä Oasiksen palvelinprosessi jumiutui tilaan, jossa se ei enää valvonut päätelaitteiden olemassaoloa vaan jätti palomuurisäännöt auki vaikka päätelaite olisikin poistunut julkisen verkon alueelta. Tämä toimintavirhe puolestaan mahdollisti helpomman yhteyden kaappaukseen perustuvan mahdollisuuden murtautua pääsynvalvojasta läpi.

On mahdollista, että huolellisella testauksella, jossa pääsynvalvojaa olisi kuormitettu samalla lailla, tämä virhe olisi huomattu aikaisemmin. Nyt kuitenkin testaus tarkoituksella unohdettiin (*Graff, Van Wyk 30: Mikä unohtui?*) vaatimattomien testausresurssien vuoksi tai ainakin siirrettiin tuotantokäytössä tapahtuvaksi testaukseksi ja niinpä virhekin ilmestyi näkyviin vasta kun järjestelmä oli tuotantokäytössä.

5.8. Arkkitehtuuri apuun

Tässä vaiheessa arkkitehtuurin suunnitteluun käytetty vaiva tuli selkeästi palkittua. Koska pääsynvalvoja oli verkkoarkkitehtuurissa arvioitu pääsynvalvontajärjestelmän heikoimmaksi ja todennäköisemmin murtuvaksi kohdaksi ja muut järjestelmät oli suunniteltu ja toteutettu se huomioiden, pääsynvalvontaohjelmiston virhe mahdollisti käytännössä hyökkääjälle vain helpon tien Internettiin sen sijaan, että huonoimmassa tapauksessa tie olisi avattu TTY:n sisäverkkoon. Näin pääsynvalvojan virhetoiminta oli lähes hyväksyttävän virhetoiminnan rajoissa ja siihen pystyttiin reagoimaan rauhallisemmin kuin, jos turvallisuudelle tärkeät järjestelmät olisivat olleet vaarassa. Tämä ei olisi ollut mahdollista, ellei arkkitehtuurisuunnittelussa olisi otettu huomioon kokonaisuutta (*Graff, Van Wyk 26: Turvaa kokonaisuus*) sen sijaan, että olisi keskitytty vain ohjelmiston turva-arkkitehtuuriin.

Arkkitehtuurisuunnitelu ei ole auttanut vain vikaan reagoinnissa vaan myös sen korjaamisessa. Vaasan ammattikorkeakoulun ylläpito toteutti Oasiksen vian ilmettyä samat rajapinnat alaspäin toteuttavan pääsynvalvontaohjelmiston, Tinon, joka oli mahdollista vaihtaa Oasiksen tilalle ilman, että alla olevaan palomuuriskriptin rajapintaan tarvittiin mitään muutoksia. Pääsynvalvojan kerrosarkkitehtuuri mahdollisti näin komponenttien joustavan vaihtamisen jopa niin suuressa määrin, että järjestelmän lähes kaikki komponentit, käyttöjärjestelmää myöten, oli mahdollista vaihtaa vähällä siirtämisvaivalla. Näin toteutettiin kuvassakin näkyvä pääsynvalvojan uusin versio autentikointijärjestelmää tuotteistamaan syntyneessä yrityksessä. Autentikointijärjestelmän käyttäjille muutos taas ei näy missään muualla kuin ehkä parantuneesssa toiminnallisuudessa.

5.9. Yhteenveto

Turva-arkkitehtuurin perusteet ja periaatteet eivät päde pelkästään ohjelmistojen tekoon vaan niitä voidaan ja pitää soveltaa myös järjestelmä- ja verkkoarkkitehtuurin suunnitteluun. Jos keskitytään vain yhden asian esimerkiksi pelkän ohjelmiston turvaamiseen, rikotaan jo Graffin ja Van Wykin sääntöä 26: Turvaa kokonaisuus. Turva-arkkitehtuuria, kuten mitään muutakaan arkkitehtuuria suunnitellessa ei voida unohtaa työn alla olevaan elementtiin liittyviä ja vaikuttavia muita elementtejä ja asioita. Monet tässäkin tekstissä esitellyistä periaatteista tuntuvat maalaisjärjen tai jo olemassa olevien ohjelmistotuotannon käytäntöjen mukaisilta. Esimerkkikin osoitti, että niitä voidaan noudatttaa tietämättä koko Graffista ja Van Wykistä mitään, mutta tärkeintä onkin, että ne muistetaan ja niitä noudatetaan arkkitehtuurisuunnittelua tehdessä.

Turva-arkkitehtuuri on turvallisuuden pohja, jonka päälle rakennetaan. Vaikutuksiltaan ja kustannuksiltaan suurimmat ja kalleimmat virheet voidaan tehdä ja välttää täällä. Suunnittelemalla arkkitehtuuri huolellisesti sitä voidaan myös käyttää työkaluna, jonka perusteella pystytään varautumaan ja myös reagoimaan tehokkaammin järjestelmästä löytyviin virheisiin. On käytännössä mahdotonta varmistua järjestelmän tai sen minkään komponentin virheettömyydestä ja itse täydellinen virheettömyyskin on mahdottomuus. Ainoa mahdollisuus on huolehtia, että virheisiin pystytään puuttumaan ja niiden vaikutukset pystytään minimoimaan korjauksen ajaksi. Tämän vuoksi pätevä turva-arkkitehtuuri on tärkeä osa minkä tahansa järjestelmän suunnittelua ja toteutusta.

Lähteet

[GraWyk03] Mark G. Graff, Kenneth R. Van Wyk: Secure Coding, Principles and Practices , O'Reilly & Associates; 1st edition, July 2003.

[TutPubAc] TUT Public Access, http://www.atm.tut.fi/tut-public-access/, 3.11.2003 12:59:19

6. Tietoturvallisuuden erityiskysymykset ohjelmiston implementointivaiheessa

Jani Kilpilinna

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

6.1. Johdanto

Parhaitenkin suunniteltu ohjelmisto voi epäonnistua, mikäli sen toteutuksessa epäonnistutaan. Lähdekoodi on viimeinen askel suunnittelussa ajettavaan ohjelmistoon, sillä siitä muodostuu suoraan ohjelma, jonka tietokone suorittaa. Tässä vaiheessa tehdyt virheet heijastuvat ajettavaan ohjelmaan ja mahdollisesti myös valmiiseen tuotteeseen, mikäli testauksessa ei havaita ongelmaa. Implementointi on siis tärkeässä roolissa ohjelmiston tietoturvallisuuden kannalta.

6.2. Mitä tietoturvallinen ohjelmointi on

Tietoturvallisen ohjelmoinnin lähtökohtana on, että vihamielinen koodi ei pääse käsiksi tietoon, jota ei ole sille tarkoitettu ja ettei se pääse tekemään ei-toivottuja toimintoja. Tämä edellyttää sitä, että järjestelmä itsessään mahdollistaa sen, että vihamielinen koodi ei saa luotettavan koodin oikeuksia tehdä asioita. Lisäksi täytyy löytää balanssi turvallisuuden ja käytettävyyden väliltä. Mikä on riittävän turvallinen?

Täydellisen tietoturvallinen koodi tarkoittaisi koodia, joka toimii niin kuin se on suunniteltu kaikissa tilanteissa. Käytännössä on kuitenkin tunnettu tosiasia, että kaikissa vähänkin suuremmissa järjestelmissä on virheitä, joita kaikkia ei testauksen rajallisessa ajassa pystytä löytämään. Näin ollen lopulliseen tuotteeseen jää vikoja. Tällöin ohjelmiston virheenhallinta on myös erittäin oleellisessa asemassa ohjelmiston tietoturvan kannalta.

Lisäksi ei riitä, että ohjelmisto läpäisee kaikki testitapaukset, vaan sen täytyy olla tehty alun alkaenkin tietoturvalliseksi. Nykyisin tuntuu olevan monesti vallalla asenne, missä asetetaan nopea ohjelmistokehitys tietoturvan edelle. Tämä on vaarallinen lähestymistapa.

Turvallisen koodin luominen vaatii taitoa, jota saa koulutuksen ja kokemuksena kautta. Lisäksi tarvitaan halua ja motivaatiota tehdä tietoturvallista koodia, sekä ympäristö, kiinnitetään huomiota ja arvostetaan ohjelmistojen laatua.

6.2.1. Tietoturvallisen ohjelmoinnin ongelmat

Suurin osa implementointivaiheen tietoturvaongelmista johtuu ohjelmoijan tekemästä virheestä. Joko osata tehdä tietoturvallista koodia tai ei yksinkertaisesti haluta nähdä vaivaa asian eteen. Myös ohjelman suorituskykyä voidaan kasvattaa tekemällä kompromisseja sen tietoturvallisuuden kannalta. Esimerkiksi eräs tekstikäsittelyohjelma jätti dokumentille varaamansa muistin tyhjentämättä, ilmeisesti koska haluttiin nopeuttaa

hieman ohjelman toimintaa. Seurauksena tallennetun dokumentin tietorakenteisiin saattoi jäädä arkaluontoista tietoa, kuten salasanoja, jotka vastaanottaja saattoi kaivaa esiin.

Minkään muun koodin toimintaan kuin omaan ei voi täysin luottaa. Kuitenkin lähes poikkeuksetta kaikki järjestelmät sisältävät muodossa tai toisessa osia, joihin on pakko pystyä luottamaan. Kääntäjässä tai käyttöjärjestelmässä piilevät virheet ovat erittäin vaikeita jäljittää. Tällaisessa tapauksessa periaatteessa täysin oikein tehty koodi saattaa toimia väärin. Esimerkiksi prosessorin mikrokoodissa olevaa virhettä on lähestulkoon mahdotonta löytää sovellustasolta käsin.

Voidaan myös luottaa liikaa testaukseen, vaikka tunnetusti paraskaan testaus ei pysty havaitsemaan kaikkia vikoja. Ohjelman toteuttaja on kuitenkin oman koodinsa asiantuntija ja vastuussa tuottamastaan koodista. Alusta asti tietoturvallisen koodin kirjoittamisen näkökohtien huomioiminen on myös kustannustehokasta, sillä testaus ja virheiden korjaus lisää kustannuksia. Lopulliseen tuotteeseen päässeet virheet voivat aiheuttaa tuhoisia ja taloudellisesti raskaita seurauksia.

Koska ohjelmoijat ovat keskeisessä asemassa tietoturvallisen koodin aikaansaamiseksi, tulee heillä olla ajantasaiset tiedot tietoturvallisen koodin kirjoittamiseksi. Ajantasaista tietoa ohjelmistojen haavoittuvuuksista ja ohjeita turvallisen koodin kirjoittamiseen saa esimerkiksi lukuisilta foorumeilta, alan kirjallisuudesta ja tutkimuksista. Avoimen lähdekoodin tutkiminen on tässä suhteessa hyödyllistä, sillä sitä on paljon ja usein se on katselmoitu usean henkilön toimesta. Täytyy tietysti muistaa, että myös huonosti toteutettua koodia on paljon avoimena, eli täysin kritiikittömästi sitä ei voi tutkia.

Ohjelmakoodin uusiokäyttö on suositeltavaa, mikäli kyseinen koodi on hyväksi havaittu, sitä voidaan tarkoitukseen soveltaa ja se on tekijänoikeudellisesti mahdollista. Erityisesti avoimen lähdekoodin lisenssillä, kuten GPL ja LGPL, varustettujen ohjelmakoodien käyttö on usein mahdotonta suljettua koodia käyttävässä kaupallisessa sovelluksessa, sillä tällöin ohjelmien lisenssit ovat yleensä ristiriidassa keskenään. Tällaisissa tapauksissa on tietysti mahdollista solmia erillinen lisenssi tekijän kanssa.

Paras tapaus tietysti on, mikäli voidaan käyttää ohjelmiston valmistajan omaa valmista koodia. Mikäli vanhoja moduuleita voidaan käyttää suoraan, voidaan moduulitestaus tältä osin mahdollisesti ohittaa kokonaan.

6.2.2. Tietoturvallisen ohjelmoinnin periaatteet

Pitää aina varautua siihen, että ohjelmiston toimintaan yritetään vaikuttaa vahingollisesti ja että pahin uhkakuva toteutuu. Suurin osa ohjelmistoista ottaa vastaan syötteitä muodossa tai toisessa. Näin niiden toimintaan pystytään vaikuttamaan ohjelman ulkopuolelta ja samalla ne aiheuttavat potentiaalisen tietoturvariskin. Puskurin ylivuoto on yksi laajalle levinnyt tietoturvaongelma. Puskurin ylivuoto tapahtuu kun ohjelma hyväksyy enemmän syötettä kun mihin on varauduttu. Tästä voi seurata esimerkiksi ohjelman kaatuminen, toimiminen väärin tai jopa mahdollisen murtautujan pääsy järjestelmään. Mikäli murtautujalla on käytössään ohjelman lähdekoodi, hän voi etsiä ohjelmasta haavoittuvuuksia, kuten saada ylivuotaneen datan pinoon ja sitä kautta suorittimelle. Näin esimerkiksi Robert Morris sai kuuluisan matonsa leviämään laajalle.

6.2.2.1.Syötteet

Puskurin ylivuodot ovat vältettävissä siivoamalla aina saatu syöte, mistä tahansa se tuleekin. Ikinä ei tule odottaa syötteen olevan jossain tietyssä muodossa, vaan on varauduttava siihen, että joku tahallisesti yrittää häiritä ohjelman toimintaa syöttämällä siihen odottamatonta dataa. Erityisesti syötteen mahtuminen sille varattuun tietorakenteeseen tulee varmistaa. Rajoissa pysyminen on erittäin tärkeää varmistaa kielissä, jotka tarjoavat matalan tason pääsyn osoitinartimetiikkaan, kuten C.

Konfiguraatiotiedostojen sisältöön ei voi luottaa sokeasti. On varmistettava, että sieltä tulee järkevää dataa. Esimerkiksi pieni virhe konfiguraatiotiedostossa voi aiheuttaa ohjelman väärän toiminnan. Tiedostoja voivat muuttaa ulkopuoliset tahot joten niiden sisällön järkevyys täytyy varmistaa kuten muutkin syötteet. Konfiguraatiotiedostot yleensä vaikuttavat ohjelman kriittisiin toimintoihin, joten niiden sisällön oikea käyttö on tärkeää ohjelmiston tietoturvan kannalta. On myös huomattava, että samaa tiedostoa ei avata useita kertoja, vaan käytetään tiedostokahvoja.

Lähes kaikista järjestelmistä löytyvät ympäristömuuttujat ovat tyypillinen tapa aiheuttaa niitä käyttävälle ohjelmalle ongelmia. Syöttämällä ei-odotettuja ympäristömuuttujia pyritään saamaan ohjelmisto toimimaan epänormaalilla tavalla. Kuten muidenkin syötteiden kanssa, ympäristömuuttujat on myös syytä pitää erityisessä tarkkailussa.

Tässä mainitut eri syötelähteet koskevat tyypillisiä ohjelmistoja, mutta lista ei luonnollisesti ole mitenkään täydellinen. Nyrkkisääntönä on, että kaikkien syötteiden kanssa tulee olla varuillaan, ja niiden käsittelyssä tulee toimia oikein.

6.2.2.2.Ohjelmakoodin ongelmat

Alustamattomat muuttujat ovat yllättävän suuri ongelma, erityisesti, mikäli ajoympäristö muuttuu. Tällainen tapaus voisi toteutua vaikkapa siirtämällä testattu, testiympäristössä toimivaksi havaittu tuote tuotantokäyttöön. Eri ympäristöissä alustamattomat muuttujat toimivat eri tavoin, toisissa muuttujien muistialue on alustettu, toisissa ne saattavat sisältää satunnaista dataa. Alustamattomien muuttujien aiheuttamat virheet ovat usein työläitä ja kalliita löytää, sillä ne voivat aiheuttaa täysin satunnaisia ongelmia satunnaisella frekvenssillä. Tämän tyyppisiin ongelmiin on kehitetty koodianalysaattoreita, jotka varoittavat potentiaalisesta vaarasta.

Samaan kategoriaan voidaan liittää myös muuttujien näkyvyydestä aiheutuvat ongelmat. Erityisen vaarallista on käyttää eri näkyvyysalueilla samannimisiä muuttujia, jolloin inhimillisen erehdyksen vaara käyttää väärää muuttujaa on suuri. Koodianalysaattorin asianmukainen käyttö paljastaa yleensä myös nämä virheet. Yleensä laatukäsikirjassa erityisesti kielletään alustamattomien sekä samannimisten muuttujien käyttö.

6.2.2.3.Tiedonkäsittely

Tiedostojen käsittelyssä tulee myös olla varovainen. Esimerkiksi tiedostoviittauksissa suoralla ja epäsuoralla viittauksella voidaan potentiaalisesti huijata ohjelmaa hakemaan tiedosto, jota ei ole tarkoitettu haettavaksi. Erään sovelluksen turvallisuusmekanismi tarkisti vain hakemistopolun alusta, että secret-hakemistoon ei viitata. Tämä estikin esimerkiksi suorat ”/secret/passwd”-viittaukset, mutta salli esimerkiksi ”/bin/../secret/passwd”-viittauksen, jolla sama tiedosto saatiin luettua.

Sama varovaisuus koskee myös epäsuoria tiedostolinkkejä ja hakupolkuja. Linkkien ja hakupolkujen avulla voidaan samaan tapaan saada ohjelma huijattua käsittelemään tiedostoa, jota sen ei pitäisi käsitellä. Linkeillä voidaan viitata aivan eri paikassa olevaan tiedostoon ja hakupolkua käytettäessä voidaan asettaa samanniminen ohjelma hakupolussa oikean ohjelman edelle.

Arkaluontoisen informaation tallennus täytyy toteuttaa siten, että siihen ei ulkopuolinen pääse käsiksi. Tällaista tietoa ovat esimerkiksi asiakastiedot, salasanat, luottokortin numerot ja www-sovelluksissa sessiotiedot. Tiedon suojaamiseen kannattaa käyttää kerrospuolustusta, eikä luottaa pelkästään esimerkiksi tiedostosuojaukseen. Yleensä kannattaa käyttää huolellisen tiedostosuojauksen lisäksi myös tiedon salausta. Tämä vaikeuttaa jo huomattavasti tiedon väärinkäyttöä, mikäli tiedostosuojaus tai tietokannan suojaus pystytään kiertämään.

6.2.3. Validointi

Koska ohjelmistovirheiltä ei voi toteutusvaiheessa täysin välttyä, täytyy keskittyä niiden minimoimiseen. Täydellisen testauksen ollessa käytännössä mahdotonta, on edullisinta tuottaa implementointivaiheessa mahdollisimman virheetöntä koodia. Virheitä voidaan löytää usein katselmoimalla koodia mahdollisimman monen asiantuntijan taholta. Miten formaalia tämä on, riippuu pitkälti koko prosessin formaliudesta. Myös erilaisia katselmointimenetelmiä on käytössä, esimerkiksi Extreme Programming-tyyppinen metodologia sopii joihinkin joustaviin projekteihin ja lyhentää huomattavasti koko ohjelmistotuotantoprosessia, mutta sen soveltuvuus suuriin projekteihin on rajallinen.

Katselmoinneissa kannattaa käyttää tarkistuslistoja, millaisia asioita etsitään. Tarkistuslistat voivat sisältää asioita, jotka täytyy ottaa huomioon toteuttamisessa, sekä tyypillisiä ja tunnettuja tietoturvaongelmia. Listat täytyy tietysti pitää ajan tasalla.

Joissain projekteissa pelkkä katselmointi on riittävä. Mikäli ohjelmisto on hyvin monimutkainen tai sitä käytetään kriittisissä sovelluksissa, kuten erilaisissa lääketieteen sovelluksissa, on syytä suorittaa riippumaton validointi ja verifiointi. Yleensä tämän suorittaa jokin ulkopuolinen taho, kuten alihankkija. Tässä prosessissa noudatetaan hyvin formaalia kaavaa ja koodi katselmoidaan rivi riviltä ja erilaisiin turvallisuusnäkökohtiin (safety) kiinnitetään erityistä huomiota.

Erilaisista erikoistuneista ohjelmistokehitys- ja laadunvarmistustyökaluista on apua katselmointia ajatellen, sillä niillä voidaan etsiä koodista mekaanisia virheitä, jotka voisivat jäädä inhimilliseltä katselmoinnilta huomaamatta. Tällaisia työkaluja ovat esimerkiksi koodianalysaattori, jolla voidaan havaita alustamattomat muuttuja ja muut vastaavat virhetilanteet. Kattavuusanalysaattorilla voidaan tutkia, mitä osia koodia suoritetaan milläkin hetkellä ja kuinka paljon. Tästä on etua myös koodin optimointiin. Debuggerilla voidaan tutkia muun muassa muuttujien ja pinon tiloja tietyllä hetkellä ja pysäyttää koodin ajo kokonaan haluttuihin paikkoihin. Näin voidaan tarkastella, ohjelman toimittaa ajon aikaisesti. Muistinhallintatyökaluista on hyötyä etsittäessä muistivuotoja ja tietorakenteiden rajojen ylityksiä.

Koodin ylläpidettävyys on oleellinen osa ohjelmiston tietoturvaa sen elinkaaren aikana. Ylläpidettävään koodiin on vaikeampi tehdä tietoturvavirheitä sitä tehtäessä ja päivitettäessä. Siksi on tärkeää noudattaa ylläpidettävän koodin tuottamiseen tarvittavia periaatteita. Yleensä nämä on määritelty laatukäsikirjassa. Tällaisia periaatteita ovat muun mu-

assa kommentointi, konventiot yms. Koodi joka on hyvin dokumentoitu ja modulaarinen on helpommin ylläpidettävä ja näin turvallisempi.

Käyttämätön koodi kannattaa poistaa, mikäli niin on varmasti turvallista tehdä. Tällöin täytyy varmistaa, ettei viittauksia kyseiseen koodiin varmasti ole olemassa. Kattavuusanalysaattorilla voi myös yrittää tutkia asiaa, mutta se ei takaa ettei viittauksia ole.

Kaikki muutokset koodiin täytyy testata ennen tuotantoon siirtymistä. On oleellista, että kaikki muutokset testataan yhtä huolellisesti, kuin koodi muutenkin testattaisiin.

6.2.4. Yleisiä virheitä

Jokainen funktio yleensä palauttaa jonkun paluuarvon, joka kertoo, onnistuiko funktion suoritus vai ei. Paluuarvon tarkastuksessa pahin virhe on, että sitä ei tehdä. Mikäli funktio ei onnistunut syystä tai toisesta ja ohjelman suoritusta jatketaan normaalisti, ovat seuraukset yleensä vakavat. Toinen virhe liittyy oletukseen paluuarvosta. Ajatellaan seuraavaa koodia:

```
Int ret = Authenticate_user();
If (ret == AUTHENTICATION_FAILED)
{
    // kirjautuminen epäonnistui
}
else
{
    // kirjautuminen onnistui
}
```

Jos edellisessä tulee jokin muu virhekoodi, kuten muistin loppuminen, ohjelma suorittaa silti sisäänkirjautumisen, eikä ole enää turvallisessa tilassa. Edellinen esimerkki pitäisi toteuttaa näin:

```
Int ret = Authenticate_user();
If (ret == AUTHENTICATION_SUCCESSFUL)
{
    // kirjautuminen onnistui
}
else
{
    // kirjautuminen epäonnistui
}
```

Tässä tapauksessa kirjautuminen tapahtuu ainoastaan, mikäli paluuarvona on onnistuminen, muutoin ei.

Riittämätön poikkeustenkäsittely aiheuttaa ohjelman hallitsemattoman kaatumisen. Tämä saattaa aiheuttaa tietoturvaongelman. Ohjelmiston täytyy pystyä ajamaan itsensä alas poikkeustilanteessa, tähän tarkoitukseen on käytettävissä poikkeukset. Missään tapauksessa core dumpia ei saa käyttää sillä se saattaa sisältää arkaluontoista tietoa.

Pseudo-random generaattorien käyttäminen esimerkiksi kryptografiassa on tuhoisaa. Useimmat tietokoneen muodostamat ”satunnaisluvut” ovat täysin tilastollisesti ennustettavia. Täysin satunnaista on vaikea muodostaa tietokoneen logiikalla, mutta tähän on käytössä muita menetelmiä. Esimerkiksi laavalampuista otetun liikkuvan kuvan on todettu olevan melkoisen satunnaista.

Autentikonnin tai salauksen toteutus väärin tai huolimattomasti aiheuttaa äärimmäisen turvallisuusongelman. Autentikointi täytyy tehdä sitä varten tehdyllä menetelmällä. Ei missään tapauksessa esimerkiksi IP-osoitteella, MAC-osoitteella tai vastaavilla, jotka eivät ole tarkoitettu autentikointiin. Kryptografian riittävän hyvin osaavia ohjelmistosuunnittelijoita on erittäin vähän. Valmiita kryptografiasovelluksia on saatavilla. Mikäli ei ole erittäin hyvää syytä tehdä oma kryptografiasovellus, älä tee sitä. On helppo olettaa että tieto on turvassa, kun se näyttää kryptatulta.

Ohjelmalla ja sen käyttäjillä täytyy olla ainoastaan riittävät oikeudet tehdä se mitä pitääkin. Esimerkiksi web-sovelluksia ei tule ajaa root-oikeuksilla. Ohjelmakoodiin sulautettuja käyttäjätietoja ei tule käyttää, sillä tämä altistaa nämä tiedot esimerkiksi reverse engineering-tekniikoiden käytölle ja niiden muuttaminen on hankalaa.

6.3. Verkko-ohjelmistojen erityiskysymyksiä

Verkko-ohjelmistoissa syötteiden tarkistaminen korostuu entisestään. Esimerkiksi http on tilaton protokolla, joten erilaisia tietoja, kuten sessiotietoja joudutaan tallettamaan paikkoihin, jossa niitä voidaan helposti muuttaa. Myös tiedonsiirto avoimessa verkossa luo lisää haasteita esimerkiksi elektronisen kaupan- ja mobiilisovelluksille.

Verkkosovellusten käyttämät http-muuttujat, evästeet ja lomakkeita tulevat tiedot ovat helposti muutettavissa tai sovellukselle voidaan yrittää syöttää täysin mielivaltaista syötettä. Avoimessa verkossa on erittäin helppoa suorittaa tahallista häirintää tai hyökkäyksiä sovelluksia vastaan. Tämän vuoksi sovellukset täytyy toteuttaa siten, ettei tällainen toiminta aiheuta varaa sen toiminnalle.

Eräs esimerkki huonosti toteutetusta parametrien tarkastuksesta on joulukuussa 2003 Internet Explorer-selaimesta löydetty tietoturvaongelma. Mikäli URL:ssa on tiettyjä näkymättömiä merkkejä, kuten 0x01, URL:n loppuosaa ei näytetä. Tämä mahdollistaa sen, että käyttäjälle voidaan helposti uskotella hänen olevan eri palvelimella kun oikeasti on. Esimerkiksi voidaan antaa seuraavanlainen linkki:
http://www.fakedomain.com@realdomain.com. Jos @-merkin eteen sijoitetaan näitä tiettyjä escape-merkkejä, käyttäjän selaimessa näkyy ainoastaan username-osa, jolloin häntä voidaan harhauttaa. Tekemällä tarpeeksi vakuuttavan näköinen sivu ja antamalla käyttäjälle sopiva linkki sivulle useiden henkilöiden tilitiedot ovat päätyneet vääriin käsiin. Selaimen valmistajan ratkaisu ongelmaan toistaiseksi on ollut varsin erikoinen: kirjoita kaikki URL:t käsin.

Mahdollisessa murtautumistilanteessa sovelluksen, erityisesti verkkokauppasovelluksen täytyy pystyä estämään järjestelmän hyväksikäyttö. Tämä tuo ongelman; pitäisikö sovelluksen sulkea tällaisessa tilanteessa kaikki kaupankäynti? Tämä saattaa aiheuttaa suuriakin tulon- ja imagonmenetyksiä ja saattaa olla häirinnän tarkoitus. Sovelluksen täytyy olla alunperinkin niin hyvin suunniteltu ja toteutettu, ettei tällaista pääsisi tapahtumaan.

Avoimen verkon yli tapahtuva kommunikaatio täytyy erityisesti suojata, jotta se ei pääd y selväkielisenä kolmannelle osapuolelle. Lisäksi täytyy pystyä varmistamaan se, että kommunikaatio todella tapahtuu kahdenvälisenä, eikä tule kolmannelta osapuolelta, ja että siirretty data säilyy muuttumattomana. Tähän tarkoitukseen on olemassa erilaisia suojausmenetelmiä.

6.3.1. Verkkosovellusten sudenkuopat

Käyttäjän syöttämää syötettä ei saa koskaan käyttää shell-parametreina. Tämä tarjoaa erittäin helpon tavan tuottaa vahinkoa järjestelmälle. Tämä koskee erityisesti CGI-ohjelmia, jotka käyttävät erillisiä ohjelmia. Esimerkiksi käyttäjä voisi helposti antaa komennon parametrin perään puolipisteen ja vahingollisen komennon, vaikkapa 'rm –rf /' aiheuttaisi ikäviä tilanteita. Vähimpien oikeuksien periaate pienentää mahdollisia tuhoja ja www-palvelimen täytyy toimia aina omilla oikeuksillaan, jotka on määritelty niin tiukoiksi kun on mahdollista. Jotkut CGI-kielet mahdollistavat käyttäjän syötteiden käyttämisen komentoriviltä kokonaan ja näitä ominaisuuksia kannattaa käyttää. Täytyy myös varmistaa, ettei käyttäjän syötteitä käytetä suoraan tietokantaan liittyvissä toimenpiteissä.

Kuten muissakin sovelluksissa, käyttäjän syötteiden tarkistuksen ja siivoamisen laiminlyönti on verkkosovellusten suurimpia riskitekijöitä. Erityisesti C- ja C++ ohjelmat ovat alttiita puskurin ylivuodoille, jolla voidaan helposti kaataa palvelin. Vain sallitut merkit tulee hyväksyä, tietotyyppien oikeellisuus pitää erikseen varmistaa. Minkäänlaiseen asiakaspuolen validointiin ei pidä luottaa, sillä asiakaspuolen sivuja on helppo muuttaa ja esimerkiksi JavaScript voidaan kytkeä pois päältä. Lisäksi käyttäjällä on hallussaan käyttäjäpuolen sivujen lähdekoodi, joka paljastaa tietoa ohjelman syötteistä.

6.4. Lähdekoodi

Palvelinpuolen lähdekoodin paljastamisessa on omat riskinsä. Mikäli tietoturva-aukkoja on jäänyt ohjelmistoon, lähdekoodi paljastaa kaikki nämä aukot julkisuuteen. Mikäli lähdekoodi on suljettu, täytyy se pitää myös turvassa katseilta. Esimerkiksi aiemmin Microsoftin Internet Information Server antoi lähdekoodin ulos, jos tiedostonimen perässä oli piste. Toinen variaatio tästä oli lisätä URL:n perään ::$DATA. Nämä ominaisuudet on korjattu, mutta korostaa palvelinohjelmistojen pitämistä ajan tasalla.

Monet tekstieditorit säilyttävät varmistustiedostoja tiedostoista. Mikäli palvelin sekä ajaa ohjelmia että jakaa staattisia tiedostoja, voidaan hyvin arvaamalla saada näitä varmistustiedostoja vääriin käsiin. Tämän vuoksi staattista sisältöä ei tule pitää samassa paikassa kuin ajettavia tiedostoja ja palvelin täytyy konfiguroida oikein.

6.5. Virhetilanteiden käsittely

Jotkut ohjelmat auttavat tarpeettomasti potentiaalista murtautujaa antamalla liikaa tietoa virhetilanteissa. Esimerkiksi epäonnistuneiden SQL-lauseiden näyttäminen ei ole kovin järkevää, sillä se paljastaa tietokannan rakennetta, joka tulisi pitää piilossa. Myös muut virhetilanteet eivät saa antaa käyttäjälle ylimääräistä tietoa tapahtuneesta; eihän normaali käyttäjä tätä tietoa tarvitse. Servleteissä kaikki poikkeukset täytyy käsitellä ja tuotantoympäristöissä tarkat virhetiedot kytkeä pois päältä.

6.6. Tietoturvallisuus

Tuotantoympäristöstä täytyy poistaa kaikki ylläpitoon liittyvät ja testiohjelmat. Jotka eivät käyttäjän saataville ole tarkoitettu. Täytyy huomioida, mitä hakemistoja palvelin oikeastaan jakaa. Esimerkiksi yleensä jakamalla jokin hakemisto, myös sen alihakemistot ovat jaossa. Hakemiston lukemisen estäminen ei missään tapauksessa riitä, hyvin arvaamalla tai brute force-hyökkäyksellä voidaan tiedostot silti lukea. Käyttämällä tunnettuja tiedostonnimiä on mahdollistettu esimerkiksi luottotietojen päätyminen vääriin käsiin.

Autentikonti täytyy tehdä turvallisesti. Esimerkiksi pelkkä perus http-autentikointi ei ole tällainen, sillä salasanat liikkuvat suojaamattomina. Useimmat verkkosovellukset eivät estä brute force-tyyppisiä hyökkäksiä, joten eri salasanojen kokeilu on helppoa myös suojatuilla yhteyksillä. Sovelluksen tulisi sulkea käyttäjäprofiili, mikäli siihen yritetään kirjautua liian monta kertaa epäonnistuneesti. Jos sama käyttäjä on kirjautuneena sisään useasta IP-osoitteesta samanaikaisesti, on tämä usein merkki paljastuneesta salasanasta. Jokaiselle käyttäjälle pitäisi antaa oma tunnus joka voidaan helposti sulkea tarvittaessa. Usein turvaudutaan antamaan jaettuja tunnuksia, mutta tämä tekee mahdottomaksi seurata, kuka on kirjautuneena ja pääsyn estämisen vaikeaksi. Omien lokien kerääminen on erittäin suotavaa, sillä palvelinohjelmistojen lokit eivät yksinään yleensä anna riittävästi informaatiota.

Palvelimen kaikki ominaisuudet, joita ei tarvita tulee sulkea. Tämä korostaa ylläpitäjän asiantuntemusta, sillä monissa palvelinohjelmistoissa monet ominaisuudet ovat oletusarvoisesti päällä. Paraskaan palomuuri ei pysty estämään hyökkäyksiä, jos niille itse avaa oven.

6.7. Johtopäätös

Tietoturvallisen ohjelmiston toteutus ei ole helppo tehtävä ja vaatii paljon resursseja. Yleiset ongelmat ja haavoittuvuudet ovat kuitenkin vältettävissä valistamalla ohjelmoijia yleisistä turvallisuutta heikentävistä virheistä, tekemällä katselmointeja ja testaamalla ohjelmistot erityisesti tietoturvallisuuden kannalta.

7. Weto-projekti tietoturvallisen ohjelmoinnin näkökulmasta

Jani Kilpilinna

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

7.1. Yleistä

Web Teaching Organiser, eli lyhyesti Weto on Tampereen yliopiston tietojenkäsittelytieteiden laitoksen projekti, jonka tarkoituksena on kehittää laitoksen ja yliopiston tarpeisiin soveltuva verkko-ohjelmisto kurssien hallintaan. Tavoitteena on tehdä tarpeisiin mukautuva joustava järjestelmä, joka tarjoaa opettajille työkalut kurssien hallintaan ja opiskelijoille yhteneväisen liittymän suoritusten palauttamiseen ja arvostelun seurantaan.

Vastaavia järjestelmiä on saatavilla runsain määrin, mutta minkään valmiin järjestelmän ei ole todettu olevan kaikilta osin kriteerit täyttävä, joten uuden järjestelmän kehittämiseen päätettiin ryhtyä.

7.2. Arkkitehtuuriratkaisut

Weto on arkkitehtuuriltaan melko tyypillinen verkkosovellus. Käyttöliittymä on toteutettu servleteillä, joten ohjelmisto on mukavasti käytettävissä standardit täyttävällä www-selaimella. Käyttöliittymä koostuu siis servletien generoimasta HTML 4.01 Transitional-koodista. Itse ohjelmalogiikka on toteutettu Javalla. Käytännössä ohjelmalogiikka hoitaa tietokantaliittymän ja käyttöliittymän kommunikoinnin keskenään. Tietokannaksi on valittu mySQL. Staattisten HTML-dokumenttien palvelemiseen on käytetty Apachea ja servleteistä huolehtii Jakarta Tomcat.

7.3. Tavoitteet

Aluksi tavoitteena on rakentaa modulaarinen ja tarpeiden mukaan kustomoitavissa oleva järjestelmä tietojenkäsittelytieteiden laitoksen käyttöön. Suunnittelussa on huomioitu koko yliopiston tarpeet mahdollisimman hyvin, joten myöhemmässä vaiheessa ohjelmisto on sovellettavissa yliopiston laajuiseen käyttöön. Tavoitteena on julkaista sovelluksen lähdekoodi, jolloin kuka tahansa halukas taho voi vapaasti käyttää sovellusta. Tämä edellyttää tietysti ohjelmiston turvallisuuden varmistamista.

7.4. Automatisointi

Projektin tarpeita varten koodin generointia on automatisoitu. Avoimeen lähdekoodiin perustuvaan Fujaba-editoriin on tehty tietokantoja muodostava ja ylläpitävä lisäosa. Näin saadaan visuaalisella työkalulla tehtyä tietokanta sekä Java-luokat ER-kaavion perusteella. Näin ollen muutokset tietokantaan ovat yksinkertaisia ja turvallisia, sillä tietokantageneraattori huolehtii osaltaan tietokannan ja sen liittymien oikeellisuudesta. Näin inhimillisen erehdyksen mahdollisuus on tältä osin minimoitu.

7.5. Sovelluksen toiminta ja tietoturvallisuus

Servleteissä tapahtuu jatkuvasti toistuvia asioita, esimerkiksi valikot muodostetaan ja sivukohtainen autentikointi hoidetaan aina periaatteessa samalla tavalla. Myös jos haluttaisiin tehdä esimerkiksi ulkoasuun tai autentikointiin muutoksia, ja käyttöliittymän sivujen muodostus tehtäisiin servlet-kohtaisesti, pitäisi muutokset tehdä jokaiseen servletiin. Tämän vuoksi on järkevää, että tällaiset toistuvat tapahtumat ovat hajautettu omaan luokkaansa. Tämä helpottaa ylläpidettävyyttä ja turvallisuutta, koska mekanismi on toteutettu vain yhdessä paikassa.

Sovelluksen tietokantakerros on hajautettu siten, että esimerkiksi tietokannan vaihtaminen on mahdollisimman yksinkertaista. Tämä parantaa myös turvallisuutta, sillä muutoksia varten tarvitsee huolehtia ainoastaan tietokantakerroksen toimivuudesta, sillä servletit käyttävät tällöinkin määriteltyä API:a ja toimivat edelleen oikein mikäli tietokantakerros toimii oikein. Tietokannan samanaikaisuudesta huolehtiminen on toteutettu ohjelmallisesti sovelluksen toimesta. Tämä pitää huolen siitä, että ohjelmisto havaitsee tilanteet, joissa joku muu on tehnyt välillä muutoksia tietokantaan. Tämä mahdollistaa kevyempien ja ilmaisten tietokantojen käytön, joissa ei ole samanaikaisuuden hallintaa valmiina. Myös nämä ominaisuudet tarjoavien tietokantojen käyttö on silti mahdollista.

Koska ohjelmistoa ajetaan Java sandboxissa, on esimerkiksi tyypilliset tietorakenteiden ylivuodot pitkälti ratkaistu. Java huolehtii muistinvarauksista dynaamisesti, joten käyttämällä sen tarjoamia palveluita, vältytään siltä, että on varattu liian vähän tilaa tarvittaville tietorakenteille.

Arkaluontoisen tiedon tallettamiseen on kiinnitetty alusta asti huomiota. Sovellus tarvitsee esimerkiksi opiskelijoista, joita ei ole soveliasta paljastaa kolmansille osapuolille. Näitä ovat esimerkiksi tiedot opinnoista, sähköpostiosoitteet yms. Tietokanta itsessään on salasanasuojattu, joten sitä ei pääse käsittelemään ilman sitä. Tämän lisäksi arkaluontoinen tieto on kryptattu, mikä vähentää tiedon käyttökelpoisuutta siinäkin tilanteessa, että tietokannan salasana pystyttäisiin kiertämään tai murtamaan.

Koska sovellus toimii avoimen verkon, tässä tapauksessa internetin välityksellä, on tiedonsiirrossa otettava huomioon turvallisuusnäkökohdat. Siirrettävän tiedon tulee olla riittävän hyvin salattua, jotta sen sisältö ei selviä kolmannelle osapuolelle. On myös kyettävä varmistamaan, että vastaanotettu tieto todella tulee siltä taholta, kun on odotettavissa. Lisäksi tiedon täytyy olla muuttumattomana, eli siinä muodossa kun se oli lähetettäessä. Tätä varten sovellus käyttää Secure Socket Layer (SSL)-tekniikkaa, eli kryptausta ja sertifikaatteja, joilla nämä asiat pystytään riittävällä tasolla varmistamaan.

Sovellus ottaa huomioon erilaiset käyttäjätasot. Luonnollisesti esimerkiksi opiskelijalla ei tule olla pääsyä kaikkien opiskelijoiden tietoihin, vaan ainoastaan omiinsa. Sovellus huolehtii siitä, että käyttäjät pääsevät käsiksi vain niihin tietoihin kun on tarpeellista. Eri käyttäjätasoja ovat pienimmistä oikeuksista suurimpaan: opiskelija, assistentti, opettaja ja ylläpitäjä. Lisäksi sovellus tarjoaa kaikille avointa olevaa tietoa, esimerkiksi kurssikuvaukset.

Vakiintuneiden tekniikoiden, kuten Javan ja servletien käyttö merkitsee sovelluksen turvallisuutta. Koska käytetyt tekniikat ovat olleet jo aiemmin laajassa käytössä, niiden suurimmat haavoittuvuudet ovat valmiiksi kenttätestattuja. Mahdolliset uudet aukot tulevat nopeasti julkisiksi, joten tämä vaatii kehitys- ja ylläpitohenkilöstöltä valppautta ja turvallisuusfoorumeiden aktiivista seuraamista. Mahdollisissa middlewaren päivitystilanteissa täytyy ottaa huomioon, että valitaan vakaa versio, eikä kokeellisia testiversioi-

ta. Koska lähdekoodi on tarkoitus julkistaa, on myös mahdollisella hyökkääjällä tiedot sovelluksen toiminnasta. Middlewaren oikea konfigurointi on keskeisessä asemassa sovelluksen tietoturvallisuuden kannalta.

Erilaisia syötteitä käytetään esimerkiksi sovelluksen toiminnan ohjaamiseen. Esimerkkinä tällaisesta on halutun sivun nimen lähettäminen http-parametrina. Koska käyttäjä voi halutessaan syöttää sovellukselle mitä haluaa, http-parametrit ja evästeet voivat sisältää mitä tahansa. Verkkosovelluksen tulee tällaisessakin tilanteessa toimia oikein. Mikäli tässä tapauksessa annettu sivu löytyy ja käyttäjällä on sinne oikeudet, sivu voidaan esittää. Varsinaisissa linkeissä on oikeat ja sallitut parametrit, joten periaatteessa ainoa tilanne, missä parametrit ovat väärin, tapahtuu käyttäjän toimesta. Tällaista viallista tietoa voi vapaasti syöttää sovellukselle, sillä mikäli käyttöoikeudet eivät ole riittävät tai sivua ei ole, annetaan tästä virheilmoitus. Näin ollen parametreja muuttamalla ei pysty aiheuttamaan haittaa sovelluksen toiminnalle.

7.6. Autentikointi

Käyttäjän autentikointi ja käyttöoikeuksien tarkistus toimii http-sessioiden avulla. Mikäli autentikointi jollain sivulla epäonnistuu, sivua ei näytetä ja sessio suljetaan virheilmoituksen kera. Sivua ei missään tapauksessa muodosteta ellei autentikointi onnistunut, esimerkiksi mikäli paluukoodiksi tulee jokin ennakoimaton koodi, autentikonnin katsotaan epäonnistuneen. Poikkeukset ovat jatkuvassa valvonnassa. Mikäli jostain aiheutuu poikkeus, sovellus sulkee session jättäen järjestelmän turvalliseen tilaan. Tietokantaan tehtävät muutokset ovat atomisia, joten tietokanta jää aina turvalliseen ja validiin tilaan.

Sovellus on tehty mahdollisimman modulaariseksi, jotta sen muokkaaminen eri tarkoitusperiin olisi mahdollisimman helppoa. Kerroksellisuus ja modulien paloittelu tekee koodista helpon ylläpitää ja myös turvallisemman, sillä implementointivirheiden todennäköisyys pienenee, mitä yksinkertaisempia yksittäiset modulit ovat. Mahdollisuuksien mukaan on käytetty hyväksi havaittuja palasia, esimerkiksi connection poolia ei ole tehty itse, vaan käytetty valmista sovellusta. Kyseisellä connection poolilla voidaan rajoittaa myös resurssien käyttöä, jotta järjestelmää ei ole mahdollista tukehduttaa varaamalla ääretöntä määrää yhteyksiä.

Vastuullisuus on varmistettu lokien käytöllä. Lokeja muodostavat sekä middleware, että sovelluskerros. Näin voidaan pitää kirjaa siitä, kuka on tehnyt mitäkin ja milloin. Tämä helpottaa myös eri tilanteiden konstruoitavuutta, mikä on tietysti ongelmatilanteiden selvittämiseen oleellista.

Ohjelmistotuotantoprosessin aikana koodia on katselmoitu käytännössä viikoittain, aina kun jotain oleellista on saatu aikaiseksi. Näin kehitysryhmällä on ollut kuva siitä, mitä on saatu aikaiseksi ja toteutus ei ole ollut vain yhden henkilön tietämyksen varassa. Toteutusta on dokumentoitu siten, että joku muu voi jatkaa kehitystä ja tietää mitä on tarkoitus tehdä ja miksi.

7.7. Validointi

Varsinaiseen validointi ja verifiointiprosessiin ei ole vielä ryhdytty, sillä se ei ole vielä ajankohtainen. Koodin oikeellisuutta implementointivaiheessa on kuitenkin pyritty varmistamaan katselmoinneilla ja koodianalysaattorin käytöllä. Ennen toteutusvaihetta on tehty käyttötapausdokumentti, joka kuvaa toteutettavat toiminnot yleiseltä tasolta.

8. Testaus osana tietoturvallista ohjelmointia

Reetta Karjalainen

<reeta.karjalainen@nokia.com>

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

8.1. Yleistä

Tehokkaan testaamisen avulla ohjelmiston tietoturvatasoa on mahdollista arvioida ja parantaa kaikissa ohjelmistokehitysprosessin eri vaiheissa. Suunnitteluvaiheessa hyödynnetään pisteytys- ja tarkistuslistoja halutun tietoturvatason varmistamiseksi ja laaditaan riski- ja vastakeinoanalyysit ohjelmiston käyttöönottoa ajatellen. Toteutusvaiheessa testaus taas tarkoittaa varsinaista tietoturvallisuuskysymyksiin keskittyvää staattisista ja dynaamista testausta. Lopuksi tutkitaan, miten koko käyttöönotetun järjestelmän tietoturvatasoa voidaan arvioida.

8.2. Tietoturvallisuuden testaaminen

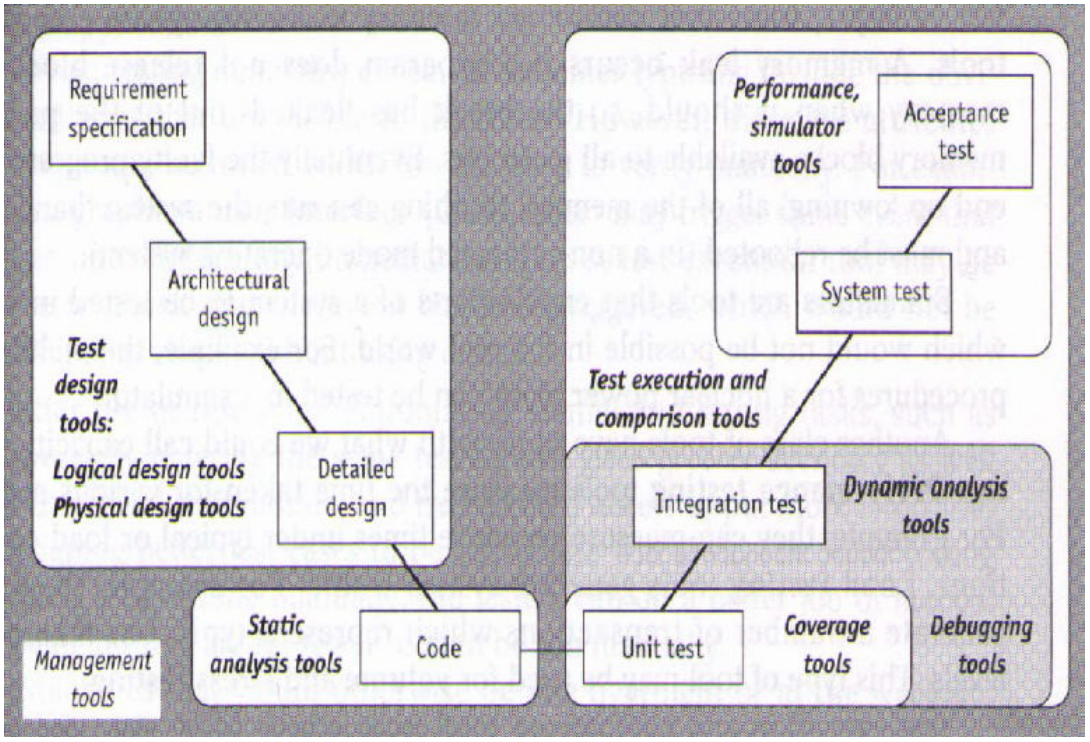
Testaus tarkoittaa analyysiä, jossa sitä, ”mitä on”, verrataan siihen, ”mitä pitäisi olla”. Testaus ei korvaa huolellista suunnittelua, mutta puutteet testauksessa näkyvät myöhemmin epämiellyttävällä tavalla tuotteen ei-toivottuna ja jopa vaarallisena toimintana. Ohjelmistokehityksen kannalta testaus on prosessi, jossa käytetään ja ylläpidetään ”testwarea” (s.o. kaikkea testaukseen liittyvää materiaalia: testaussuunnitelmia testauskoodia, testidataa jne.), tarkoituksena mitata ja parantaa testattavan ohjelmiston laatua. [Craig ja Jaskiel, 2002].

Tietoturvaa sivuavat erityiskysymykset (kuten esim. käyttäjän tunnistaminen ja tiedon salaaminen) otetaan testauksessa usein huomioon vasta tuoteprosessin loppumetreillä järjestelmätestausta suunniteltaessa arvioitaessa ohjelmiston vika- ja kuormitussietoisuutta, suoritus- ja toipumiskykyä sekä käyttäytymistä virhetilanteissa. Turvallisuus on kuitenkin myös laadukkuutta; siksi turvallisuustestauksen pitäisi koostua sekä normaalin toiminnallisuuden testauksesta että testauksesta kaikkia tunnettuja uhkakuvia vastaan, ja kuten testauksen yleensäkin, se tulisi ulottua kaikkiin ohjelmistokehitysprosessin vaiheisiin.

Ihannetapauksessa testausympäristö olisi aina valmis ja ajan tasalla, ilman minkäänlaisia muutoksia testipetiin, toimintatapoihin tai testitapauksiin! Käytännössä testausprosessiin liittyviä ongelmia ovat hitaus, manuaalisen työn suuri määrä sekä testauksen vaihteleva laatu. Testien mahdollisimman hyvä kattavuus ja suorituksen helppous ovat usein keskenään ristiriitaisia vaatimuksia, mutta molempia halutaan. Testauksen automatisoinnilla vältytään tekemästä sama työtä moneen kertaan, vaikka käytettäisiinkin useita eri testausmenetelmiä. [Heikkinen et al.,. 1998]

Tietoturvallisuuden testaamisesta on lisäksi huomattava, että toisin kuin yleistä sovelluksen toiminnan oikeellisuutta arvioitaessa, turvatakuut ovat aina voimassa vain tois-

taiseksi. Ajan mittaan tehdyt muutoksen käytettävässä ohjelmistossa tai sen ympäristössä voivat altistaa turvallisuusriskeille. Näiden turvallisuuteen liittyvien uhkakuvien tiedostamisen ja kirjaamisen pitäisi olla hyvä motivaattori tehostetulle testaukselle; ongelmana eivät enää ole pelkästään virukset tai madot, vaan järjestelmän väärinkäyttöön ja hajottamiseen tähtäävät hyökkäysmetodit.



Kuva 15. Testausmenetelmät ohjelmistokehitysprosessin eri vaiheissa [Fewster ja Graham, 1999].

8.3. Testaus suunnitteluvaiheessa

Mitä varhaisemmassa vaiheessa virheet löydetään, sitä pienemmäksi jäävät korjaamiseen kuluva aika ja siitä sekä virheellisestä toiminnasta aiheutuvat kustannukset. Etenkin massatuotteissa ohjelmiston päivittäminen käyttäjille on kallis, joskus jopa suorastaan mahdoton operaatio, eikä se enää edes paranna tuotteen tai valmistajan kuvaa käyttäjän silmissä.

Suunnitteluvaiheessa tehtävä testaaminen poikkeaa olennaisesti myöhemmästä varsinaisesta testaustoiminnasta. koska mitään testattavaa ohjelmakoodia ei vielä ole - tarkoituksena on siis ainoastaan käydä läpi design-dokumentaatiota. Suunnitelmien läpi käyminen on tehokas mutta yksinkertainen laadunvarmistusmetodi: Paakin [2001] mukaan erään arvion mukaan tarkastukset ja katselmoinnit löytävät n. 60% kaikista virheistä. Lisäksi erilaisten käyttötilanteiden mukaan suunnatut katselmoinnit löytävät vielä 35 % enemmän virheitä kuin koko suunnitelman kattamaan pyrkivät tarkastukset.

Katselmoinnissa suunnitteludokumenteista olisi tutkittava järjestelmän arkkitehtuurinen ilme, ja se miten, jako komponentteihin on toteutettu. Tietoturvallisuuden näkökulmasta tärkeää on myös komponenttien kapselointi, ts. se, millä yksiköillä on oikeudet sovelluksen kriittisiin tietoihin, sekä miten eri komponentit kommunikoivat keskenään ja ympäristön kanssa. Suunniteltu toiminnallisuus olisi paras esittää selkeinä vuokaaviona, ja päätöksen-tekopuiden pitäisi olla arvioitavana ainakin päätasolla.

Tietoturvatasoa tulisi aina pyrkiä parantamaan siellä, missä se järjestelmän puitteissa on mahdollista – tällöin myös ohjelmiston käyttöympäristöä tulisi ottaa arvioinnin kohteeksi. Jos esim. käyttöjärjestelmä on jo valittu, tietoturva voi edelleen parantaa valitsemalla systeemiin jonkun yleisesti käytetyn ja tunnetusti vakaaksi ja vikasietoiseksi havaitun kommunikointiprotokollan [Graff ja Van Wyk, 2003] Myös tulevaan testaamistyöhön tulisi varautua laatimalla ja katselmoimalla testaussuunnitelma myöhempiä vaiheita varten.

Testaaminen edellyttää aina paitsi ammattitaitoa, myös halua löytää virheitä. Lisäksi itse ohjelmiston laatu voi asettaa rajoja tulevalle testaamiselle. Friedmanin ja Voasin [1995] mukaan ohjelman testattavuudella mitataan ohjelmiston kykyä paljastaa omat virheensä ohjelmakoodissa, komponenteissa tai vaatimuksissa – ts. sitä, miten läpinäkyvä järjestelmä on. Pystyttäessä näyttämään suurella varmuudella toteen, että ohjelmassa olevat viat ovat ilmiselviä ja tulevat testauksessa esiin, kyetään myös osoittamaan vikojen puuttuminen ja ohjelman toiminnan oikeellisuus varmemmin ja nopeammin. Hyvin suunnitellut järjestelmän osat voidaan siis verifioida testaamalla, kun taas huono suunnittelu kostautuu tarpeena käyttää muita menetelmiä, esim. analysointia tai simulointia. Testattavuutta voidaan huomattavasti parantaa helpottamalla ohjelman ymmärtämistä, ja suunnitteluvaiheessa tämä on vielä suhteellisen helppoa ja nopeaa.

Käyttämällä suunnitelmien kuvaamisessa jotakin formaalia menetelmää (esim. graafispohjaista kieltä, kuten OMT, UML tai SDL) parannetaan olennaisesti järjestelmän testattavuutta. Tällöin on periaatteessa mahdollista tehdä jo pelkille suunnittelukaavioille semanttista analysointia esim. loogisten porsaanreikien ja iki-silmikoiden varalta. Yhteiskäyttöisillä, graafisilla suunnittelutyökaluilla (esim. UML- ja SDL-editoreilla) myös tehdyn työn näkyvyys kaikille suunnittelijoille paranee, vähentäen virheiden todennäköisyyttä.

8.3.1. Pisteytys- ja tarkistuslistat

Tarkistuslistojen käyttö katselmoinneissa on suositeltavaa, mutta tyypillisesti toimivatkin listat ovat voimakkaasti systeemiriippuvaisia – niiden kehittäminen edellyttää siis pitkää kokemusta kohdeympäristössä toimimisesta, eivätkä ne juurikaan ole sovellettavissa toisiin ohjelmointikohteisiin. Tarkistuslistat usein myös keskittyvät vain implementointisääntöihin,

Hyviin suunnitteluperiaatteisin nojautuville tarkistuslistoille olisi kuitenkin tarvetta enenevässä määrin jo nyt ja tulevaisuudessa. Kähkipuron [2000] mukaan komponenttitekniikkaprojektien epäonnistumisen syynä on usein sekä kokemuksen että toimivien suunnitteluprosessien puute; tekniikkaa kehitetään suurella vauhdilla markkinoille huolellisen harkinnan kustannuksella. Kuitenkin huolellisella ja oikeaksi varmennetulla suunnittelulla rakennetut ohjelmistot - erityisesti valmiit ohjelmistokomponentit – olisivat huomattavasti pitkäkäyttöisempiä, ja siten myös kaupallisesti merkityksellisempiä.

Tietoturvakysymyksissä erilaisista pisteytyslistoista voi kuitenkin saada huomattavaa hyötyä oikein sovellettuna, ja ne myös ohjaavat ohjelmiston suunnittelua oikeaan suuntaan pakottaessaan suunnittelijat ottamaan huomioon tietoturvakysymykset jo varhaisessa vaiheessa. Eräs yleisesti saatavilla oleva yleiskäyttöinen tietoturvan pisteytyslista on SAG (l. ”Security At a Glance”), joka listaa 20 ohjelmiston käyttöön ja yrityksen tietoturvakäytöntöihin liittyvää kysymystä. Testistä saatu pistemäärä ei sinänsä kerro paljonkaan ohjelmiston turvatasosta, vaan pisteytystietojen hyödyntäminen edellyttää ver-

tailukohtaa, esim. ohjelmiston tietoturvatasoa voi yrittää verrata muiden käytössä olevien ohjelmistojen saamiin tuloksiin [Graff ja Van Wyk, 2003].

	KYLLÄ	EI
1.) Pahimmassakin systeemin tietomurto- tai toimimattomuustapauksessa kustannukset < N milj. $?	+5p	0 / -5p
2.) Systeemin jäljellä oleva käyttöikä (HUOM! EI suunniteltu-) < X kuukautta?	+5p	0 / -5p
3.) Systeemillä on < Y käyttäjää?	+4p	0 / -4p
4.) Systeemin käytölle on olemassa erityiset turvallisuusohjeet ja ne on annettu tiedoksi sen käyttäjille ja ylläpitäjille?	+3p	0 / -3p
5.) Systeemin käyttäjät ja heille määritellyt käyttöoikeudet tarkastetaan kuukausittain?	+2p	0 / -2p
6.) Systeemi on suojattu käyttäjäkohtaisilla salasanoilla	+0p	0p

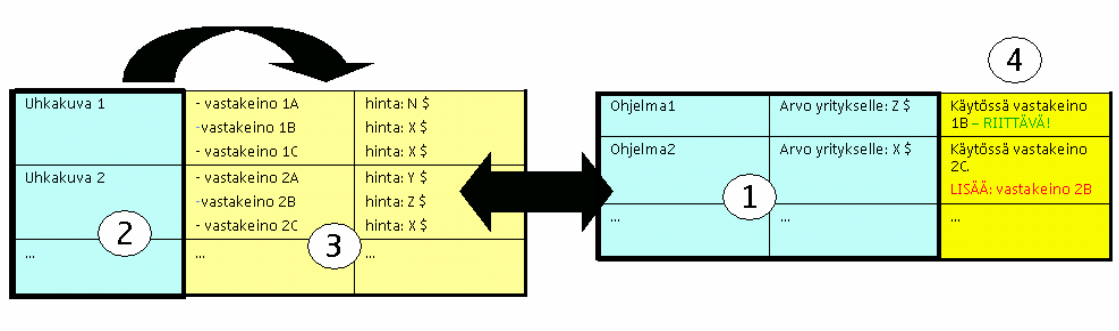
Kuva 16. Esimerkki SAG:in kysymyksistä ja pisteytyksestä [Graff ja Van Wyk, 2003].

8.3.2. Riski- ja vastakeinoanalyysit

Riskianalyyseissa lähdetään yleensä siitä realistisesta lähtökohdasta, että koska täydellistä turvatasoa on mahdotonta saavuttaa, on järkevää pyrkiä selvittämään todellisen, riittävän suojautumistason tarve. Keskeistä tällaisissa analyyseissa on sekä suojattavien kohteiden että mahdollisten uhkakuvien priorisointi; riskit voidaan jakaa ohjelman tietoturvakriittisyyden mukaan esim. luokkiin ”ei hyväksyttävä” (s.o. ehdottomasti fataalit riskit), ”siedettävä” (s.o. riski voidaan hyväksyä jos sen toteutumisen todennäköisyyttä voidaan riittävästi pienentää) ja ”hyväksyttävä” (s.o. vaikutuksiltaan merkityksettömät riskit) [Somerville, 2000]. Selvityksen ensimmäisessä osassa tyypillisesti arvioidaan, mitkä sovellukset ovat (taloudellisesti) tärkeitä yritykselle, ja keskitytään sitten tutkimaan niiden toimintojen turvaamista.

Analyysin seuraavassa vaiheessa laaditaan selvitys yleisistä tietoturvauhkista sekä niiden taloudellisista vaikutuksista (s.o. ”worst case”-skenaario). Selvitykseen lisätään sitten ehdotukset mahdollista vastakeinoista tietoturvauhkia vastaan, sekä arviot niiden implementointikustannuksista. Kun mahdollisia turvakeinoja sitten verrataan merkityksellisimmiksi katsottujen ohjelmistojen kohdalla jo toteutettuihin ratkaisuihin, saadaan valmis, listaus korjaustarpeista – tärkeysjärjestyksessä ja kustannuksineen.

Suurempien järjestelmien riskianalyyseja voidaan myös automatisoida, ja tällaisia valmiisiin kysymyslistoihin ja tunnettuihin tietoturvallisuuden uhkakuviin perustavia kaupallisia työkaluja on jo saatavilla. Vaikka osa ohjelmistotaloja uhkaavista turvallisuusriskeistä saattaakin olla luonteeltaan liian ympäristösidonnaisia, jotta valmisohjelmistot niitä kaikkia kunnolla tavoittaisivat, tällaisista ohjelmistot voivat auttaa tiedonkeruun koordinoinnissa yrityksen sisällä [Graff ja Van Wyk, 2003].



Kuva 17. Uhkakuviin, tuotteiden arvoon ja nykyiseen suojaustasoon pohjautuva riskianalyysi.

8.4. Testaus toteutusvaiheessa

Dynaamisessa testauksessa arvioidaan järjestelmän toimivuutta, sekä tutkitaan sitä tunnettujen implementointiongelmien (esim. puskureiden ylivuotojen) ja toisaalta tunnettujen tietoturvauhkien varalta. Tässä testausvaiheessa komponenttien yksikkötestaus luo pohjan koko ohjelmiston testaukselle. Regressiotestauksella varmistetaan komponentin/kerroksen toimivuus kaikkine ominaisuuksineen muutostöiden jälkeen. Integrointitestauksessa ei enää testata kerroksien sisäistä toiminnallisuutta, vaan kerrosten välitä kommunikointia ja yhteistoimintaa. Tyypillisesti yksikkötestauksen paljastamat virheet ovat ohjelmointivirheitä, kun taas integrointitestauksessa esiin tulevat virheet johtuvat suunnitteluvirheistä (yl. yhteensopivuus- tai ajoitusongelmia rajapinnoissa). Järjestelmätestauksen ongelmat johtuvat useimmiten jo vaatimusmäärittelytasolla tapahtuneista virheitä, ts. ohjelmisto ei toimikaan halutulla tavalla, vaikka toiminnallisuus vastaakin spesifikaatioita.

Joissakin tapauksissa riittävän testikattavuuden saavuttaminen voi olla ongelmallista järjestelmän huonon testattavuuden ja suuren ehdollisen (s.o. liputetun) koodin määrän takia. Tilatestaus on eräs tapa tehdä kohtuullisen kattavaa yksikkötestausta suhteellisen pienille ohjelmistokomponenteille silloin kun muut kattavuusmetriikat eivät vaikuta mielekkäiltä. Komponentin tila määritellään tällöin kaikkien niiden tietojen kombinaationa, jotka vaikuttavat komponentin käyttäytymiseen. Tällöinkin testausympäristön asettamat aika- ja muistirajat voivat osoittautua ongelmaksi – etenkään integrointitestauksessa ei useinkaan ole mahdollista saavuttaa tilakattavuutta tilojen suuren määrän vuoksi. Ottamalla simulointi avuksi voidaan käydä läpi vain testauksen kannalta oleelliset osat toiminnallisuudesta, ja sopivalla testauskielellä (esim. TTCN) toteutettu testaus voidaan automatisoida.[Heikkinen et al.,. 1998]

Sulautettujenkin ohjelmistojen testaaminen kehitysympäristössä ennen kohdeympäristöön siirtymistä on suositeltavaa, ja usein houkuttelevaakin työasematestauksen parempien työkalujen (esim. debuggereiden, muistin testaajien ja kääntäjien), takia vaikka tulokset voivat suurestikin poiketa kohdeympäristössä tapahtuvasta testauksesta. Työasemaympäristö on myös helpompi jakaa muiden testaajien kanssa, mikä mahdollistaa ryhmä- ja parityöskentelyn (”eXtreme programming) ja tehokkaan automatisoinnin kehittämisen.

8.4.1. Staattinen testaaminen

Ohjelmiston testausta ilman, että ohjelmakoodia varsinaisesti suoritetaan, kutsutaan staattiseksi testaukseksi. Koodin analysointi voidaan suorittaa joko käsin tai automaattisesti staattisen analyysin työkaluilla. Menetelmällä pyritään löytämään koodista loogisia, koodaus- ja suunnitteluvirheitä. Virkasen [2002] mukaan automaattiset menetelmät eivät tyypillisesti tuota yhtä hyvää tulosta, koska systeemin tuntevat suunnittelijat voivat tehokkaasti keskittyä etsimään juuri tietyn tyyppisiä todennäköisiä virheitä.

Ohjelmointiympäristöissä kääntäjät toimivat virheenjäljitystyökaluina. Tällaiset automaattiset analysaattorit tutkivat ja etsivät tyypillisesti funktiokutsujen parametrien tyypit, parametrien käyttöä, paluuarvoja, samannimisten muuttujien (uudelleen) määrittelyjä, suorittamatonta ohjelmakoodia, muuttujien käyttöä ja esittelyä, funktioiden käyttöä, sijoitettavia arvoja sekä osoittimien ja indeksien käyttöä (esim. viittaukset määriteltyjen. rajojen ulkopuolelle).

Uno

- Etsii käytettyjä, alustamattomia muuttujia sekä alustamattomia ja muistialueen ulkopuolille osoittavia osoittimia (“**U**nitilialized variables, **N**il-pointers and **O**ut-of-bound arrays indexing”) C-kielisistä ohjelmakooditiedostoista,

ks. cm.bell-labs.com/cm/cd/who/gerard .

RATS

- Tutkii C, C++, Perl, Python ja PHP kooditiedostoja tunnettujen turvallisuusongelmien varalta,

ks. www.securesotfware.com/download_rats.htm .

Kuva 18. ”Static code checkers”

8.4.2. Dynaaminen testaaminen

Komponentin tai ohjelmiston evaluointia itse ohjelman tai sen suorituksen aikana kutsutaan dynaamiseksi testaamiseksi. Dynaamisen testaamisen apuna voidaan käyttää dynaamisia virheenjäljitysohjelmia, l. ”debuggereita”, joilla voidaan hallitusta seurata esim. muuttujien arvojen muuttumista, aliohjelmakutsuja, ja joskus myös muistin sisältöä ohjelmaa käytettäessä. Yksinkertaisimmillaan arvioinnissa on kyse kattavuudesta, t.s. tulevatko kaikki ohjelmat polut/tilat käytyä läpi. Perinteiseen, funktionaaliseen tapaan toteutetussa ohjelmistossa tällainen testaus on varsin suoraviivaista, mutta olioita hyödyntävissä ohjelmissa komponenttien hierarkia on usein hämärtänyt, eikä selkeitä seurattavia suorituspolkua enää ole, jolloin muut testausmenetelmät on tällöin merkityksellisempiä [Somerville, 2000].

Muut virheenjäljitysohjelmat toimivat usein ns. suodatuskerroksena testattavan ohjelman ja ympäristön välillä. Tällöin ne tyypillisesti ne ottavat vastaan järjestelmä kutsut, parsivat ne, ja arvioivat niiden oikeellisuuden ennen kuin päästävät viestit eteenpäin. Osa virheenjäljitystyökaluista taas on erikoistunut muistin hallitaan tai tietynlaisten tietorakenteiden oikeellisuuden varmistamiseen.

PurifyPlus

- Etsii mahd. muistovuotoja ajetusta ohjelmistosta,

ks. www.rational.com/products/pqc(index.jsp .

ISS Database Scanner

- Käy läpi tietokantoja (MS-SQL, Sybase, Oracle) tunnettujen yleisten määrittelyvirheiden varalta, ks. www.iss.net .

Kuva 19. ”Runtime code checkers”

Profilointiohjelmien käyttäminen edellyttävää, että systeemin oikea toimintatapa on määritelty (joko suunnitteludokumenteissa tai suoraan ohjelmallisesti toiminnasta), ja tämän jälkeen systeemin toimintaa tarkkaillaan poikkeuksien varalta ohjelmiston normaalin käytön aikana, jolloin systeemin epätyypillinen toiminta saattaisi viitata mm. virushyökkäykseen.

Yksinkertaisten nauhoitus-ja-toisto -testiohjelmat (l. ”record-and-playback”) ovat helppokäyttöisiä ja toistettavuudeltaan hyviä, mutta niiden heikkous on siinä, että ne kertovat, onko toiminta saman laita kuin ennenkin – eivät siis kykene arvioi toiminnan tai tulosten oikeellisuutta sinänsä. Mikäli testattava järjestelmä muuttuu liiaksi, tällaiset testit menettävät myöskin referenssinsä ja siten käyttökelpoisuutensa esim. regressiotestauksessa [Fewster ja Graham, 1999].

Papillon

- Havainnoi ja varoittaa käyttäjien mahd. hyökkäyksistä järjestelmää vastaan, ks. www.roge.org/papillon .

Janus

- Rajoittaa tiettyjen sovellusten tekemiä järjestelmäkutsuja,

ks. www.cs.berkeley.edu/~daw/janus/ .

Kuva 20. ”Profiling tools”

Käytännöllinen menetelmä on myös järjestelmän testaaminen käyttämällä virheellisesti toimivia komponentteja (”fault injection”). Tällöin saadaan paitsi kuva siitä, miten järjestelmä kykenee toimimaan viallisesta komponentista huolimatta (s.o. viansietoisuus).

8.4.3. Järjestelmän tietoturvatason testaaminen

Ennen ohjelmiston tuotantoon päästämistä koko järjestelmä pitää testata kokonaisuutena. Siinä missä ohjelmakoodin toimintaa voidaan tutkia debuggereiden avulla, voidaan tietoverkko-protokollien analysaattoreilla tehdä vastaavaa tarkastelua ylemmällä tasolla. Kaiken verkossa tapahtuvan kommunikaation tarkastelu ja verifiointi on työläs tehtävä, mutta se on äärimmäinen keino varmistaa että oikeassa käyttötilanteessa tapahtuvat ainakin kaikki oikeat asiat.

Vähemmän kriittisten ohjelmistojen kohdalla olennaisinta taas on tunnistaa ja löytää virheet, jotka varmasti johtavat vaaratilanteeseen. Tietoturvatasoa voidaan tällöin arvioida ns. uhkakuviin perustuvilla turvallisuusargumenteilla, jolloin ei tarvitse todista koko ohjelman oikeaa toimintaa, vaan ainoastaan se, että turvallisuuden keskeisesti liittyvät osat toimivat oikein [Somerville, 2000]. Näin saavutettua tietoturvatasoa voidaan arvioida kokeilemalla erilaisia tunkeutumistestejä käyttöjärjestelmän tai tietoverkon tasolla. Tarkoituksena on tutkia, ovatko järjestelmän hyödyntämät tietoverkkoyhteydet suojattu ulkopuolisilta, asiattomilta sisäänpyrkijöiltä.

Hyökkäyksentunnistusohjelmistoja ei yleensä ehkä lasketa testaustyökaluihin, mutta niiden avulla voidaan huomattavasti parantaa ohjelmiston tietoturvatasoa. Varoitusjärjestelmiä on usein kritisoitu siitä, että ne aiheuttavat monia perusteettomia hälytyksiä. Tämän kiertäminen on kiinni lähinnä vain järjestelmän ylläpitäjien asiantuntemuksesta, sillä useat saatavilla olevat ohjelmistot tarjoavat mahdollisuuden muokata hälytysjärjestelmää juuri todennäköisimpiä uhkakuvia (”attack graphs”, esim. miten tunkeutuja voisi yrittää päästä muuttamaan hyväksyttyjen käyttäjätunnuksien listoja) vastaaviksi. [Graff ja Van Wyk, 2003]

Nmap - Yleisesti käytetty porttiskanneri, ks. www.nmap.org .   ISS Internet Scanner - Tutkii järjestelmän haavoittuvuutta tietoverkkotasolla, kykenee myös tunnistamaan hyökkäyksiä, ks. www.iss.net .

Kuva 21. Penetration testing tools”

8.5. Tietoturvatestauksen tulevaisuus

Siinä missä ohjelmiston yleistä käyttöturvallisuutta heikentävät vahingot ovat yleensä tahattomia, vakavat tietoturvallisuuden loukkaukset ovat yleensä tahallisia ja suunniteltuja [Somerville, 2000]. Testaajien toimenkuva näyttää siis tulevaisuudessakin varsin turvatulta ammatilta; toisaalta monimutkaistuvat systeemit mahdollistavat aivan uusien hyökkäysmetodien kokeilemisen, ja toisaalta taas jatkuvasti lisääntyvässä ohjelmakoo-

dissa jo piileskelevät turvallisuusriskit odottavat vain paljastumistaan. Testattavan materiaalin määrä siis kasvaa, ja lisäksi myös välttämättömiä testiskenaarioita on aina vain enemmän – eikä niiden automatisointikaan onnistune ihan automaattisesti.

Lisäksi ohjelmistoyrityksen näyttävät edelleen poistavan virheitä mieluummin testaamalla kuin tarkastamalla, katselmoimalla ja oikeaksi todistamalla. Paakin [2001] mukaan esimerkiksi ohjelmistoyritysten suvereenilla ykkösellä, Microsoftilla, on palveluksessaan yhtä paljon testaajia kuin ohjelmoijia. Tällaisessa kehitysprosessissa ei sitten juuri muuta tapahdukaan kuin koodausta ja testausta: ohjelmoijat tekevät päivisin virheitä, joita testaajat sitten öisin urakalla etsivät. Usko siihen, että ”laatu” hoidetaan vielä kuntoon testaajien joukkovoimalla, on kova.

Ohjelmistojen käyttötilanteissa käyttäjälle lankeaa usein käytännön testaajan rooli – ja tämän pätee myös tietoturvakysymyksiinkin. Keskustelujen seuraaminen tietoliikenteeseen ja tietotekniikkaan liittyvillä julkisilla foorumeilla on siis tärkeää. Vaikka yritykset itse kertovat tuotteidensa puutteista ja tarjoavat päivityksiä, erityisesti yleisesti käytetyistä ohjelmistotuotteista löytyy kaiken aikaa turva-aukkoja, jotka voivat edellyttää käyttäjiltä suojaustoimenpiteitä jo ennen korjauspaketin käyttöönottoa. Riittävän tietoturvallisuustason ylläpito edellyttää siis aktiivista kuuntelua ja ilkeää mielikuvitusta – kauhukuvia vastaan pitää osata varautua jo ennakolta.

Lähteet

[Craig ja Jaskiel, 2002] R.D.Craig ja S.P.Jaskiel, *Systematic Software Testing*. Artech House Publishing, 2002.

[Fewster ja Graham, 1999] Mark Fewster ja Dorothy Graham, *Software Testing Automation: Effective Use of Test Execution Tools*. Addison-Wesley, 1999.

[Friedman ja Voas, 1995] Michael A. Friedman ja Jeffrey M. Voas, *Software Assesment: Reliability, Safety, Testability.* John Wiley & Sons, Inc, 1995.

[Graff ja Van Wyk, 2003] Mark G. Graff ja Kenneth R. Van Wyk, *Secure Coding: Principles and Practices*, luku 6: Automation and Testing. O’Reilly & Associates, 2003.

[Heikkinen *et al*.,. 1998] Jouni Heikkinen, Jukka Korhonen ja Tero Manninen, Sulautettujen ohjelmistojen testaus: Testausko vaikeaa? *Prosessori*, maaliskuu (1998), s. 93-95.

[Kähkipuro, 2000] Pekka Kähkipuro, Komponenttiarkkitehtuurien vaikutus ohjelmistotuotantoon. *Systeemityö*, 1 (2000), s. 2-5.

[Paakki, 2001] Jukka Paakki, Turha testaaminen on turhaa. *Systeemityö*, 1 (2001), s. 10-12.

[Somerville, 2000] Ian Somerville, *Software Engineering (6th Edition).* Addison-Wesley, 2000, 6. painos

[Virkanen, 2002] Hannu Virkanen, Ohjelmistojen testaus ja virheenjäljitys. *Pro Gradu-tutkielma*, Kuopion yliopisto, 2002

9. Testauksen ja validoinnin automatisointi

Reetta Karjalainen

<reeta.karjalainen@nokia.com>

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

9.1. Johdanto

Järkevällä automatisoinnilla voidaan tehostaa ohjelmistokehitysprosessia sekä toteutus- että testausvaiheessa – mutta miten automatisointia käytetään on, mitä sillä voi saada aikaan, ja mitä se tarkoittaa ohjelmiston tietoturvallisuuden kannalta? Ja miten automaattisesti tuotetun ohjelmistokoodin oikeellisuudesta voi varmistua – automatisoidustiko?

9.2. Miksi automatisoida?

Virheiden etsiminen ja korjaaminen ohjelmakoodista aiheuttaa ylimääräisiä kustannuksia ohjelmistoprojektille. Lisäksi ohjelmistojen koon jatkuvasti kasvaessa ja komponenttitekniikoiden yleistyessä on tullut ilmeiseksi, että kaikkien virheiden korjaaminen koko niiden levinneisyysalueelle jälkikäteen on tolkuttoman kallista ja paikoin jopa mahdotonta. Mikäli ohjelmiston tulevaa käyttötarkoitusta - ja siksi myös ja vaadittavaa tietoturvatasoa - on vaikea etukäteen määritellä, virheiden tehokas poistaminen ohjelmistoista etukäteen on erityisen ratkaisevaa.

Kun yritysten tavoitteena on tuotekehityssyklien lyhentämien ja ohjelmistojen helppo muokattavuus uusiin ympäristöihin, ja kun toisaalta on otettava huomioon myös uusien toimintaympäristöjen turvattomuus, ohjelmistotuotannon kehitys- ja ylläpitotyön pullonkaulaksi on muodostumassa testaus. Paitsi että testaaminen hidastaa ohjelmiston kehitysaikataulua, se vaatii myös paljon työvoimaa: arviot testauksen osuudesta vaihtelevat 30-50 prosentin välillä suhteessa koko ohjelmiston kehittämiseen tarvittavasta työpanoksesta. Suuri osuus selittyy osaksi sillä, että testauksessa manuaalisesta tehtävän työn osuus voi ympäristöstä riippuen olla merkittävä [Heikkinen *et al.*,. 1998].

Automatisoinnilla pyritään karsimaan inhimillisten virheiden mahdollisuutta ohjelmistokehitysprosessin eri vaiheissa. Testauksessa automatisointi lisäksi säästää tekemästä sama työtä moneen kertaan, ja testisuunnitelmia ”komponentoimalla” niiden – kuten testattavien ohjelmistojenkin - uudelleenkäytettävyyttä voidaan lisätä merkittävästi. Automatisoinnin avulla voidaan myös ottaa laajempaa käyttöön testit, joiden suoritus manuaalisesti olisi hidasta ja raskasta, kuten esim. GUI-testaus tai stabiliteetti- ja stressitestaus todellisessa verkkoympäristössä.

Ohjelmistojen toteutuksessakaan automatisointi ei ole niin uusi asia kuin mitä äkkiseltään voisi tuntua - kääntäjät ovat tavallaan automatisoinnin ensimmäinen askel, joka tosin nykyään ei riitä enää kovin pitkälle. Implementointivaiheessa automatisoinnin lupailemia etuja ovat ajan säästö sekä ohjelmakoodin oikeellisuus (ts. tarkka vastaavuus suunnitelmiin.), joka osaltaan parantaa myös ohjelmiston testattavuutta. Etenkin tietoturvakriittisten ohjelmien verifiointi on kallista ja aikaa vievää, ja saattaa muodostaa

jopa yli 50% koko ohjelmistoprojektin kustannuksista [Somerville, 2000]. Tällöin formaalien suunnittelumenetelmien käyttäminen ja automatisoitu verifiointi voivat tuoda merkittäviä säästäjä.

Automatisoidun ohjelmistotuotannon toimiessa tarkoitetulla tavalla hyvin suunniteltu ohjelmisto olisi siis jo enemmän kuin puoliksi tehty. Riittävän yleisellä mutta eksaktilla tasolla kertaalleen tehty ohjelmiston toiminnallisuuden määrittely voi myös osoittautua hyvinkin monikäyttöiseksi, jos siitä voitaisiin erilaisia koodingenerointi työkaluja käyttämällä tuottaa kulloisenkin kohdeympäristön kieli- ym. vaatimusten mukainen ohjelma. Toteutusvaiheen automatisointi voisi myös säästää enemmän aikaa välttämättömälle testausvaiheelle.

9.2.1. Implementoinnin automatisointi

Generoimalla koodia automaattisesti suoraan suunnitteludokumenteista vältytään virheiltä, joita ihminen todennäköisesti tekisi kirjoittaessaan tuhansia samankaltaisia koodirivejä. Yksinkertaisimmillaan generointi voisi tarkoittaa vain käytettävien luokkien määrittelyjen tuottamista automaattisesti esim. keskinäiseen hierarkiaan asetetuista taulukoista. Seuraavilla kehitystasoilla generointi voisi tarkoittaa luokkien tietojäsenten sekä niiden saantifunktiot muodostamista, sekä edelleen. olioiden funktioiden luomista luokkakaavioiden pohjalta. Vaativimmalla tasolla ohjelmakoodia voitaisiin generoida jopa suoraan ohjelmiston halutun toiminnallisuuden kuvauksista [Kangas, 2000].

Keskeistä automatisoidussa ohjelmistotuotannossa on järjestelmän huolellinen suunnittelu; koska koodi pitäisi ehkä voida generoida design-dokumenteista monella eri menetelmällä, tarvitaan joustava ja selkeä arkkitehtuuri, joka mahdollistaa eri tapojen hyödyntämisen nyt ja tulevaisuudessa [Kylmäkoski, 2002]. Suunnitteludokumentaatio itsessään voi olla käyttötarpeesta riippuen joko graafisessa tai tekstuaalisessa muodossa. CASE-välineitä (Computer Assisted Software Engineering) suositaan niiden tyypillisesti tarjoaman graafisen näkymän vuoksi, mutta joissakin tapauksissa myös yksinkertainen pseudokoodi voi toimia sekä virallisena dokumentaationa että generoinnin alustana.

9.2.2. Ohjelmakoodin generointi pseudokoodista

Silloin kun suunnitellussa luokkahierarkiassa on suuri määrä rinnakkaisia, perusteiltaan samantapaisia luokkia, altistaa luokkakaavioiden muuttaminen tekstipohjaisista kuvauksista kaavioiksi inhimillisille virheille sekä tiedonkeruu- että piirtämisvaiheessakin. Luontevaa olisi pyrkiä käyttämään jo suunnittelutasolla asioiden kuvaamiseen pseudokoodia, jolloin sen ja halutun ohjelmointikielen välinen ero on sitten myöhemmin mahdollista kuroa umpeen automatisoidusta, esim. jonkin skriptikielen avulla.

Pseudokoodimenetelmässä on lopultakin kyseessä vain tekstimuotoisen tiedon keräämisestä, käsittelemisestä ja uudelleen tulostamisesta, ja sitä voisi hyödyntää esim. kahden erilaisen systeemin välisen rajapinnan viestimääritysten tuottamiseen. Yksinkertainen tapa toteuttaa tällä menetelmällä esim. luokkakaavioita olisi kuvata halutut oliot ensin Microsoft taulukkoina, ja sitten muuntaa nämä ASCII-muotoisiksi ja tuottaa sitten (itse tehdyllä ohjelmalla) taulukoista pseudokoodi ja muokata sitä edelleen esim. C++-kielen mukaiseksi) [Kangas, 2000].

Tämä generointitapa edellyttää pilkun tarkasti toteutettuja suunnitteludokumentteja: tyypillisiä vaatimuksia ovat mm. avainsanojen erillisyys ja oikeinkirjoitus – ja joskus

myös luokkien esittelyjärjestys. Dokumentaatio ei mielellään saisi sisältää viittauksia muualle, koska ne saatettaisiin tulkita kommenttien sijaan muuttujiksi, ja numeraalien käyttäminen on luonnollisestikin pakollista arvojen määrittelyssä. Koodigeneraattorin olisi myös osattava nimetä taulukkojen yms. alkiot mielekkäästi, sekä ohittaa sisällysluettelon tapaiset kommenttisivut.

Mikä yksinkertaisempi generoitava kokonaisuus on, sitä todennäköisemmin generaattori ei tee virheitä. Toimittaessa luokkien määrittelyn tasolla tehdyt virheet ovat todennäköisesti syntaktisia; mikäli tuotoskoodi menee kääntäjästä läpi, sen virheettömyyteen voi siis kohtuullisessa määrin luottaa.

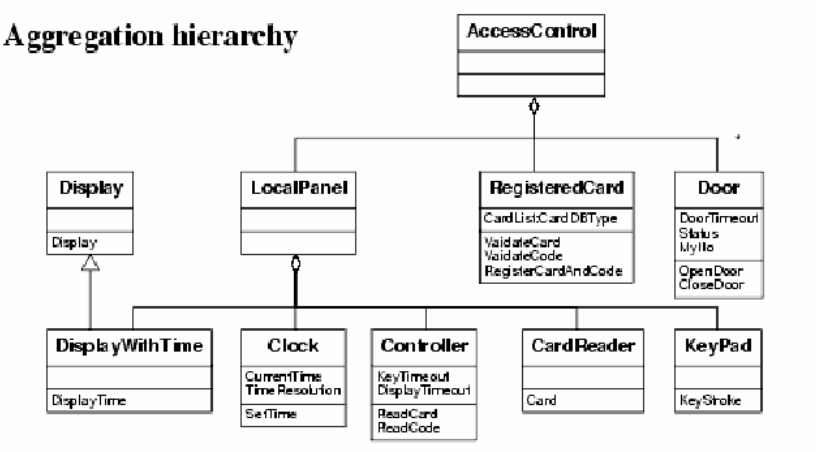
9.2.3. CASE-välineet

Markkinoilla on paljon CASE -välineitä, jolla voidaan piirrellä luokkakaavioita (jollakin oliosuunnittelunotaatiolla, kuten OMT:lla) ja generoida ohjelmakoodia tämän pohjalta. Koodin generointi on periaatteessa yksinkertaista silloin, kun se generoidaan ainoastaan kerran ja oliomallin pohjalta (s.o. staattinen malli), Tällöin luodaan vain eräänlainen runko muulle implementoinnille.

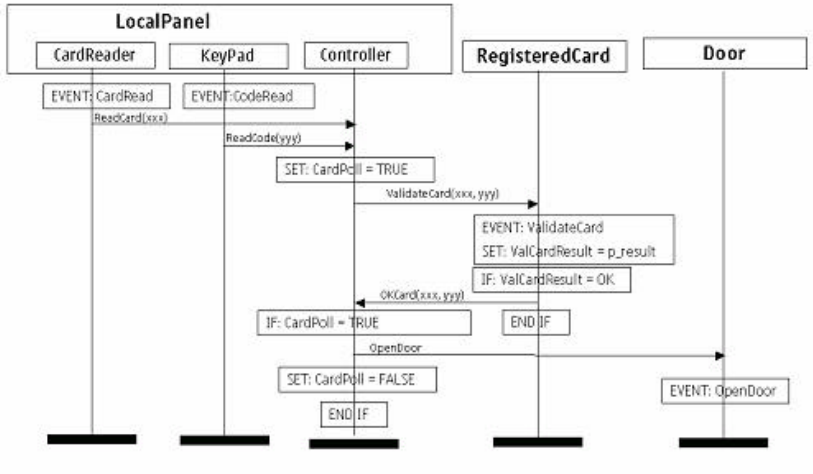
Joillakin CASE -työvälineillä voidaan luoda myös dynaamista koodia erilaisten tilakoneiden ja kulkukaavioiden pohjalta (esim. Rational Rose ja WithClass99). Tällöin syötteenä toimivat taulukkojen ohella myös ohjelmistoa suunniteltaessa syntyneet kulkukaaviot (kuvattuna esim UML-vuokaavioina tai MSC-kuvina). Eräiden näkemysten mukaan kaiken ohjelmakoodin automaattiseen tuottamiseen on turha edes pyrkiä; koska algoritmien kuvaaminen ”pseudokoodina” edellyttäisi joka tapauksessa uuden ohjelmointikielen kehittämistä, joten yhtä hyvin ohjelmiston dynaamisen osan implementoinnin voi toteuttaa jo jollakin olemassa olevalla ohjelmointikielellä [Rumbaugh, 1997]. Kuitenkin juuri dynaamisten mallien suora hyödyntäminen voisi todella vielä ohjelmistotuotannon prosesseja eteenpäin [Kylmäkoski, 2002].

Automatisoinnin vastustuksen takana voi olla myös ajatus siitä, että ohjelmistotuotteen designin ei ylipäätään katsota tarpeelliseksi sisältää tarkkaa kuvausta ohjelman kaikesta toiminnallisuudesta kaikissa tapauksissa, vaan on tyydytty kuvaamaan asiat yleisellä vaatimusmäärittelytasolla, esim. use-caseinä. Kuitenkin jo tietoturvallisuussyistä järjestelmän toiminta tulisi kuvata mahdollisimman tarkasti sen dokumentaatioon; vaatimuksissa olisi hyvä hahmottaa myös se, miten järjestelmä ei missään nimessä saisi toimia, tai millaisia virhe- ja vaaratilanteita ei saisi aiheutua. Tietoturvatason riskianalyysin tekeminen edellyttää myös kokonaisvaltaista kuvaa ohjelmiston toimintaympäristön uhkakuvista.

Muutamia automaattisten C ja C++ -koodigeneraattoreiden hyväksymiä kaavionotaatioita ovat SDL ja OMT++, joita käyttämällä voi siis yhdistää projektissa sekä suunnittelu että implementointivaiheen työn. OMT++:lla systeemi mallinnetaan OMT -luokkahierarkiakuvien ja muokattujen MCS -kaavioiden avulla; tässä viestijärjestyskaavioihin on lisätty mahdollisuus silmikoihin ja muihin ehtolausekkeisiin (ks. esimerkkikaaviot, kuvat 1 [Telelogic, 2001] ja 2). Kaavioiden sisään on myös kuvattu ns. proseduureina komponenttien toimintaa (esim. muuttujien arvojen asetus ja funktiokutsut).



Kuva 22. Esimerkki: suunnitellun kulunhallintajärjestelmän osat ja suhteet.

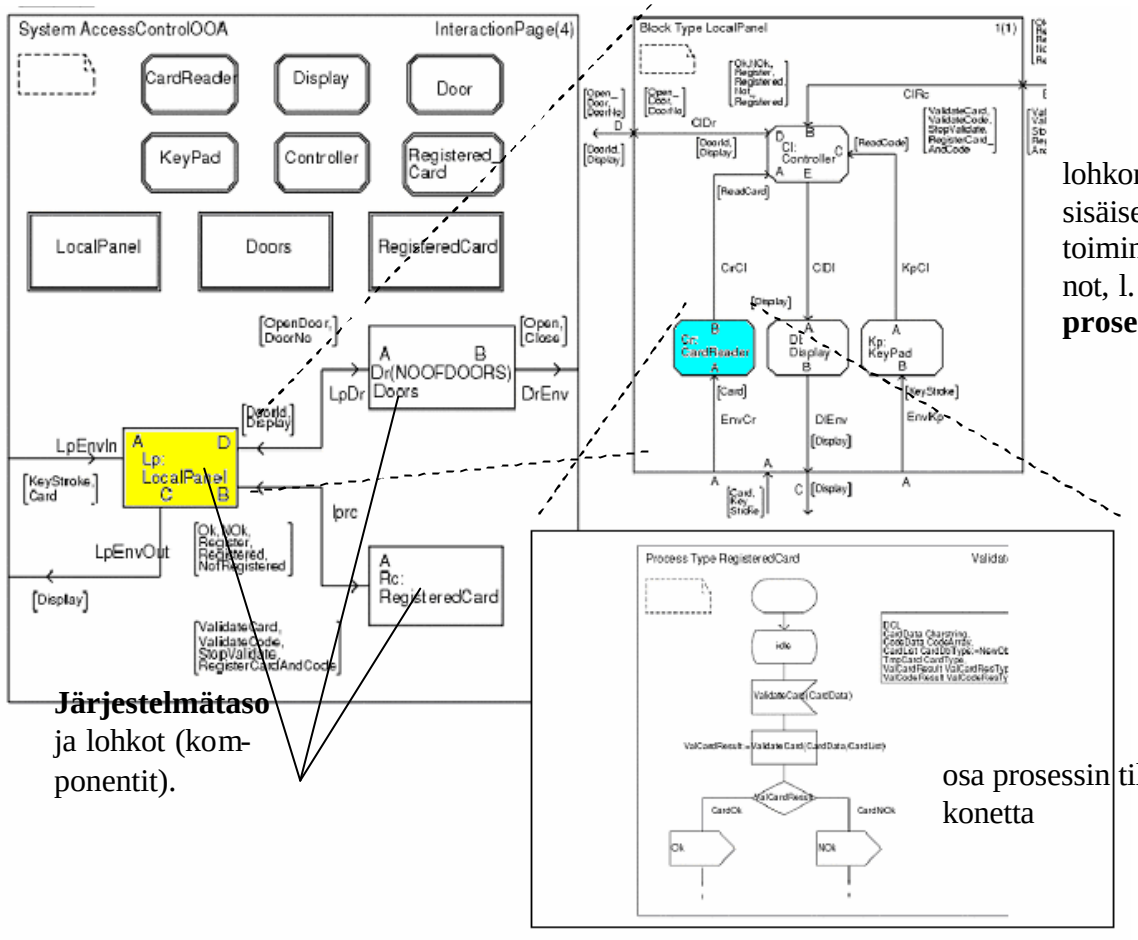


Kuva 23. Esimerkki: Oven avaus kulunhallintajärjestelmässä, notaationa OMT++.

ETSI:n nykyään ensisijaisesti käyttämä tietoliikenneprotokollien, kuvaustekniikka SDL (”Specification and Design Language”) taas määrittelee systeemin toiminnan (laajennetun, äärellisen) tilakoneen kautta. SDL on kehitetty erityisesti vuorovaikutteisille reaaliaikajärjestelmille, ja se perustuu hierarkkisiin signaaleilla kommunikoiviin oliotyyppisiin komponentteihin (l. hierarkinen black-box –malli), joten se on luonnostaan oliosuuntautunut, ja siksi houkutteleva graafinen toteutusvaihtoehto olioystävällisiin ympäristöihin.

SDL:ssä järjestelmä jaetaan päätasolla lohkokomponentteihin, joiden sisään systeemin toiminnot yleisillä tasolla rajataan itsenäiseksi tilakoneiksi eli prosesseiksi. Prosessin, eli tilakoneen, sisällä kuvataan kaikki mahdolliset tilasiirtymät UML:n tapaiselle kielellä. Koko järjestelmän toiminta koostuu näin irrallisten tilakoneiden yhteistoiminnasta [Ellsberger *et al.*,1997].

SDL on suosittu kuvauskieli, koska se on selkeän rakenteellinen ja helppo oppia. Se myös pakottaa designin selkeään arkkitehtuuriin ja määrittelemään tiukasti rajapinnat komponenteille; prosessien (l. tilakoneiden) välillä on viestikanavat ja niillä voivat kulkea vain rajatun nimisen ja muotoiset signaalit. Virheellisen muotoiset viestit eivät pääse vastaanottaja-prosessin viestijonoon. SDL ei ehkä sinänsä suojaa huonolta suunnittelulta, mutta voi kyllä tehokkaasti paljastaa sen.



Kuva 24. Kulunhallintajärjestelmä SDL-kuvina [Telelogic, 2001].

SDL-editorit tyypillisesti tarjoavat mahdollisuuden kuvatun järjestelmän syntaktiseen ja semanttiseen tarkasteluun. Syntaktinen analyysi tarkistelee kaavioita ja niissä määriteltyjä muuttujia kieliopin pohjalta. Semanttisessa vertailussa taas tutkitaan kuvattujen prosessien mielekkyyttä, esim. muuttujien vertailua – kyseessä on siis eräänlainen kääntäjä kaaviokuville. Myös prosessien sisäinen logiikka arvioidaan, esim. useampaa kuin yhtä viestiä ei voida vastaanottoa ilman tilasiirtymää, ja ehtolauseilla täytyy olla aina vähintään kaksi mahdollista, toisensa poissulkevaa vaihtoehtoa, jne.

Graafisten tilakone-lähdekoodien etuna voi pitää sitä, että niiden toiminta useimmiten generoituu pitkiksi switch-case -tyyppisiksi rakenteiksi; vaikka ne siis määrittelisivätkin vain halutun toiminnallisuuden, virhetilanne ei sinänsä aiheuta ongelmia, vaan toiminta vain pysähtyy. Useilla SDL -simulaattoreilla voidaan siksi myös tehdä simuloida vasta yleisluontoisesti määritellyn systeemin toimintaa pelkkiä signaalinnimiä käyttäen, ja seurata samalla SDL-kaavioista viestien käsittelyn etenemisestä virhetilanteiden löytämiseksi [Telelogic, 2001].

9.3. Sulautettujen järjestelmien erityispiirteitä

Sulautetuissa järjestelmissä muistin ja prosessointitehon määrä voi olla hyvin rajoitettu, mikä valitun arkkitehtuurimallin lisäksi vaikuttaa automaattisesti tuotetun ohjelmakoodin kielen valintaan. SDL saattaa edelleen kelvata systeemin kuvauskieleksi, mutta koodin generoinnissa pitää osata ottaa huomioon esim. se, jos järjestelmä ei salli olio-

ohjelmointiin perustuvia rakenteita, tai ”broadcastingia” systeemin prosessien välillä. Kaikkien muuttujien alustaminen ei ole aina järkevää tai mahdollista, mikä altistaa virheille hyvästä suunnittelusta huolimatta. Usein ei myöskään ole mahdollista kuvata kaikkia tilasiirtymiä, joten kuvaukset jäävät vaillinaisiksi, ja automaattinen semanttinen analyysi voi antaa valtavasti virheilmoituksia – tai analyysi jää kesken [Lisko, 2002]. Kuvattu toiminnallisuus saattaa myös olla vaillinaista jos koodia generoidaan ehdollisesti, ts. vain jonkun liputetun, testattavan ominaisuuden mukaan.

On huomattava, että useat sulautettujen järjestelmien kohdekääntäjät on suunniteltu ensisijaisesti käsinkirjoitetun koodin käsittelyyn – nyt esim. C-kielinen generaattori voi luoda ”switch-case” -rakenteen, jossa on satoja case-tapauksia – vaikka rakenne sinänsä olisi C-kielen mukainen, ei gcc-C-kääntäjä ehkä osaakaan sitä käsitellä. Lisäksi kääntäjissä voi olla raja käsiteltävien tiedostojen koolle, jolloin koodin generointia pitää pystyä hajauttamaan. useampaan kohdetiedostoon Sulautetuissa järjestelmissä SDL-mallien (yms.) suurin hyöty liittyy siis suunnittelun selkiyttämiseen ja varhaiseen prototypointiin kehitysympäristössä, ja on enemmänkin siten työkalu– kuin kielilähtöistä.

Mikäli varsinaista ohjelmakoodia generoidaan käytettäväksi asti, ovat valmiiseen koodin liitettävät C-kieliset lisä- ja korvausosat hyvin yleisiä. Paitsi että SDL:stä generoitu koodi ei välttämättä ole kaikilta osin riittävän optimoitua (sisältäen esim. raskaita matemaattisia laskelmia tai pitkiä silmukoita), usein on myös tarpeen käyttää jo olemassa olevia komponentteja systeemissä. Nämä osat muodostavat erityisen turvallisuus- ja toiminnallisuusriskin, koska toisin kuin kaaviokuvien, niiden automaattinen analysointi ei tyypillisesti onnistu.

9.4. Automatisoinnin ohjelmistokehityksen hyödyt ja hankaluudet

Käytetty automaattisen koodingeneroinnin menetelmä riippuu pitkälti sovelluskohteesta, eikä aiheesta ole juurikaan saatavilla yleistä teoriaa, vaan lähinnä tutkittavia työkalutoteutuksia. CASE-työkalujen, kuten SDL-sovellusten, etuja on sekä arkkitehtuuri että järjestelmän toiminnan läpinäkyvyys, jolloin suunnittelussa voidaan hyödyntää erilaisia täydellisyysanalyyseja. Mallit tarjoavat myös toteutusturvallisuutta, koska odottamatonta dataa ei hyväksytä, käytettyihin tietotyyppeihin on usein rakennettu sisään rajoituksia, ja kaikki prosessien väliset viittaukset ja kutsut ovat näkyvissä (esim. referenssilistoina tai tapahtumalokeina).

Graafien käsittelyn hankaluutena voi pitää sitä, että niiden esittely vie paljon tilaa – tilakoneet voivat käytännössä olla useita sivuja pitkiä, jolloin niitä täytyy yhdistellä ”goto”lauseilla tms. Osa tilasiirtymistä saattaa myös (esim. optimointisyistä) näyttää epädeterministisiltä tai spontaaneilta: jos kaikkien tilasiirtymien esittelyyn ei ole mahdollisuutta, käyttöön voidaan ottaa *-tiloja ja *-viestejä, jotka siis tarkoittavat mitä hyvänsä tilaa tai viestiä [Ellsberger *et al.*,1997].

Järjestelmän ei-toivottua toiminnallisuutta ei useinkaan kannata esitellä kaavioissa (esim. else- vaihtoehtoina), vaan niiden esittely jätetään muuhun dokumentaatioon. Halutun toiminnan kuvaamisen lisäksi järjestelmiin on yhä itse lisättävä poikkeustilat käsittelevät kontrollit, ja pysähtymistilanteita varten tarvitaan jonkinlainen etenemissuunnitelma. Graafeista ei myöskään ole apua muistin tai ajastinten käsittelyyn, vaan se on yhä riippuvaista toteutusalusta. Tilanteesta riippuu, mikä esim. väärän muotoiselle (järjestelmän ulkopuolelta) lähetetylle viestille tulisi tapahtua; tuhotaanko viesti (jolloin kosketaan tuntemattomaan määrään muistia) vaiko ignoroidaanko se vain (jolloin riskinä on se että muisti täyttyy ”roskasta”).

Kuvalliseen esitykseen perustuvat suunnittelutyökalut eivät tyypillisesti myöskään sovellu suurta laskentatehoa vaativiin ohjelmistoihin (esim. säänsimulaatioihin), eivätkä ne sinänsä mahdollista ohjelmiston suorituskyvyn mittausta [Telelogic, 2001]. Tällöin osaa generoidusta koodista joudutaan ehkä muokkaamaan käsin, ja vaikka muutoksien tekeminen (usein lukukelvottomaan) koodin onnistuisikin, muutokset eivät tule näkyviin dokumentaatiossa tai analyysissa, ja ne katoavat seuraavien generointien yhteydessä. Ongelmallista voi olla myös käsinkirjoitetun koodin yhdistäminen generoituun koodin silloin, kun generaattorin muille osille antama dynaaminen nimistö (esim. olioille, prosessi-id:lle jne.) vaihtelee. Mikäli työkalu sallii samojen nimien käytön eri komponenttien sisällä, koodingenerointi voi olla myös virheellistä; ympäristöä kuvaavien etuliitteiden lisääminen komponentteihin taas voi tehdä koodista täydellisen lukukelvotonta ihmissilmälle [Lisko, 2002].

Mikäli koodigeneraattori ei ole riittävän älykäs ja muokattavissa, ei generointikaan voi olla kovin omatoimista, ja tällöin säästynyt implementointiaika menee itse generoinnin suunnitteluun ja työkaluihin liittyvien hankaluuksien ennakointiin ja selvittelyyn. Dynaamisen koodingeneroinnin toimiessa edes kohtuullisesti sen etuna voi pitää sitä, suunnittelu keskittyy nyt toteutusyksityiskohtien sijaan itse systeemin rakenteeseen ja toiminnallisuuteen. Koska onnistunut koodin generointi edellyttää täydellistä designia, tulevaisuudessa suunnittelu on varmasti vaihe, jonka tulee kehittyä eniten.

SDL:n, kuten muidenkin formaaliin määrittelyyn perustuvien menetelmien, heikkoutena voidaan pitää sitä, että formaali verifiointi sinänsä ei takaa, että järjestelmä olisi käyttötarkoitukseensa sopiva, koska todellista käyttäjien näkökulma ei näy. Toisekseen, mikäli järjestelmää ei käytetä ennakoidulla tavalla, formaalista tutkinnasta ei ole mitään hyötyä [Somerville, 2000]. Koska tietoturvahyökkäykset usein perustuvat ennakoimattomaan toimintaan, on kyseenalaista, voidaanko formaaleille menetelmillä suojautua niitä vastaan – järjestelmän haluttu toiminnallisuutta niillä kyllä voidaan verifioida.

Formaalien menetelmien haittana on myös yleensä pidetty sitä, että ne edellyttävät paitsi sovellusalueen, myös itse käytettyjen menetelmien syvällistä tuntemusta, ja tällöin tuotoksien verifiointi voi vaikeutua ratkaisevasti; tietoturvallinen ohjelmistotuotanto edellyttää siis sekä tietoturvallisuuden asiantuntijuutta että prosessiosaamista. Formaaleja menetelmiä on usein myös pidetty kalliina verrattuna muihin tekniikoihin [Somerville, 2000]. Ilman verifiointimenetelmien järkevää kohdentamista (esim. erityisesti tietoturvakriittisten komponenttien toiminnan varmentamiseen) ja ilman tehokasta (automatisoituja) testausympäristöä tyypillisesti epälineaarisesti kasvavat kustannukset uhkaavat muuttua hallitsemattomaksi peikoksi, joka ahmii koko projektin resurssit.

9.5. Testauksen automatisointi

Ratkeamaton ”Turingin koneen pysähtymisongelma” todistaa, että täydellinen testaaminen on mahdotonta, ts. ei voida rakentaa tietokoneohjelmaa, joka testaa, pysähtyykö sille syötteenä annettu toinen tietokoneohjelma kaikilla mahdollisilla syötteillä [Luukkainen, 2002]. Perusongelma ohjelmistojen oikeellisuuden toteamisessa on siis se, että tavallinen ohjelmointikieli yhdistettynä vapaaseen dynaamiseen muistinkäyttöön on liian ilmaisuvoimainen, että ohjelmien oikeellisuus voitaisiin todistaa aukottomasti.

Testaaminen on siis rajoitettava tasolle, jolla voidaan olla kohtuullisen varmoja siitä, että toiminnallisuus on halutun kaltaista. Tilatestaus on yksi tapa tehdä kohtuullisen kattavaa testausta suhteellisen pienille ohjelmistokomponenteille – isommissa järjestelmissä haasteena on tilojen suunnaton määrä, koska tila täytyy määritellä kaikkien siihen vaikuttavien mahdollisten muuttujien yhdistelmäksi. Toisaalta keskittäminen tiettyihin

ongelma-aluesiin on tärkeää, koska häiriöt aja virheet syntyvät yleisimmin niissä kohdissa järjestelmää, joita käytetään useimmin – on mahdollista, että 60 % ohjelmiston virheistä saadaan korjattua, mutta että käyttövarmuus paranee ainoastaan 3%. [Somerville, 2000] Tämä pätee erityisesti tietoturvallisuutta parantamaan pyrkivään testaukseen, jolloin tehokas suojautuminen tiettyjä käytännön käyttötilanteiden uhkakuvia vastaan on tärkeämpää kuin koko toiminnallisuuden virheettömyyden verifiointi.

Käytännön käyttötilanteessa järjestelmän toiminta voidaan verifioida erilaisilla stressitesteillä: esim. maksimitestauksessa tutkitaan, mikä on maksimaalinen kuormitus tietyssä ajassa ennen kuin järjestelmä ylikuormittuu ja kaatuu (esim. kb/s, transaktiota/s jne.). Stabiliteettitestauksella taas tutkitaan ohjelmiston käyttövakautta, eli kun käytössä on normaaliksi arvioitu kuormitus, tutkitaan vasteaikojen kehitystä ja resurssien kulutusta, transaktioiden onnistuneisuutta tms. pidemmän ajan (esim. viikon) kuluessa [Mansikkamäki 2001].

Testien mahdollisimman hyvä kattavuus ja suorituksen helppous ovat usein keskenään ristiriitaisia vaatimuksia, etenkin kun tietoturvallisuussyistä kaikenlaisen testaamisen määrä monenlaisissa ympäristöissä vain lisääntyy Ihannetapauksessa kaiken tällaisen testaamisen voisi toteuttaa automatisoidusti, ilman ihmisen valvonta, ja testaamista voisi tehdä tehokkaasti ilman muutoksia testipetiin, toimintatapoihin tai testitapauksiin testausympäristöstä riippumatta.

9.5.1. Mitä voi testauksesta automatisoida?

Puoliautomaattisessa tuottamisessa testitapaukset johdetaan jostakin toisesta esitystavasta (kuten SDL:stä TTCN:iin). Täysin automaattinen testitapausten tuottaminen perustuu jonkin algoritmin mukaiseen testattavan tila-avaruuden läpikäyntiin – tässäkin tapauksessa ihmisen apua voidaan tarvita rajoittamaan tila-avaruus mielekkään kokoiseksi. Automaattisilla tilatestereillä voidaan saada testiaineiston ylläpito-ongelmaa pienemmäksi, mutta usein testitulosten analysointi jää edelleen testaajan harteille.

Testitapausten laatiminen on monesti testausprosessin aikaa vievin osio, varsinkin jos muut vaiheet on jo automatisoitu. Monissa tapauksissa testitapausten automaattinen generointi edes osittain on mahdotonta, sillä se edellyttää testattavan ohjelman toiminnan formaalia spesifiointia. Formaalista spesifioinnista on hyötyä myös testidatan määrittelyssä silloinkin kun itse testitapauksien generointi ei (esim. ei käyttötilanneorientoituneen testaussuunnittelua vaan kuormittavuustestausta) takia ole mahdollista [Mäkelä, 2000]. Toisaalta testidatan määrittelyä voidaan myös automatisoida; koska usein ohjelmat reagoivat saman kaltaisiin syötteisiin samalla tavalla, syötejoukkoja (esim. keskimääräiset arvot ja raja- tai päätearvot) voidaan tunnistamisen jälkeen generoida täysin automaattisesti.

Testipetigeneraattorit luovat testattavalle moduulille automaattisesti testipedin, jolle voidaan kuvata testikuvauskielellä (esim. TTCN) suoritettavat testitilanteet. Kielellä voidaan myös määritellä halutut testitulokset (esim. moduulin palauttamat viestit), jolloin testien ajo ja tarkastelu voidaan kokonaisuudessaan automatisoida. Pelkkä nauhoitustyökalujen käyttö (s.o. tallennetaan ajo ja verrataan seuraavia kertoja siihen) ei kuitenkaan kerro paljon ohjelmiston toiminnan oikeellisuudesta, eikä sopeudu toiminnallisuuden muutoksiin, vaan regressiohyöty menetetään.

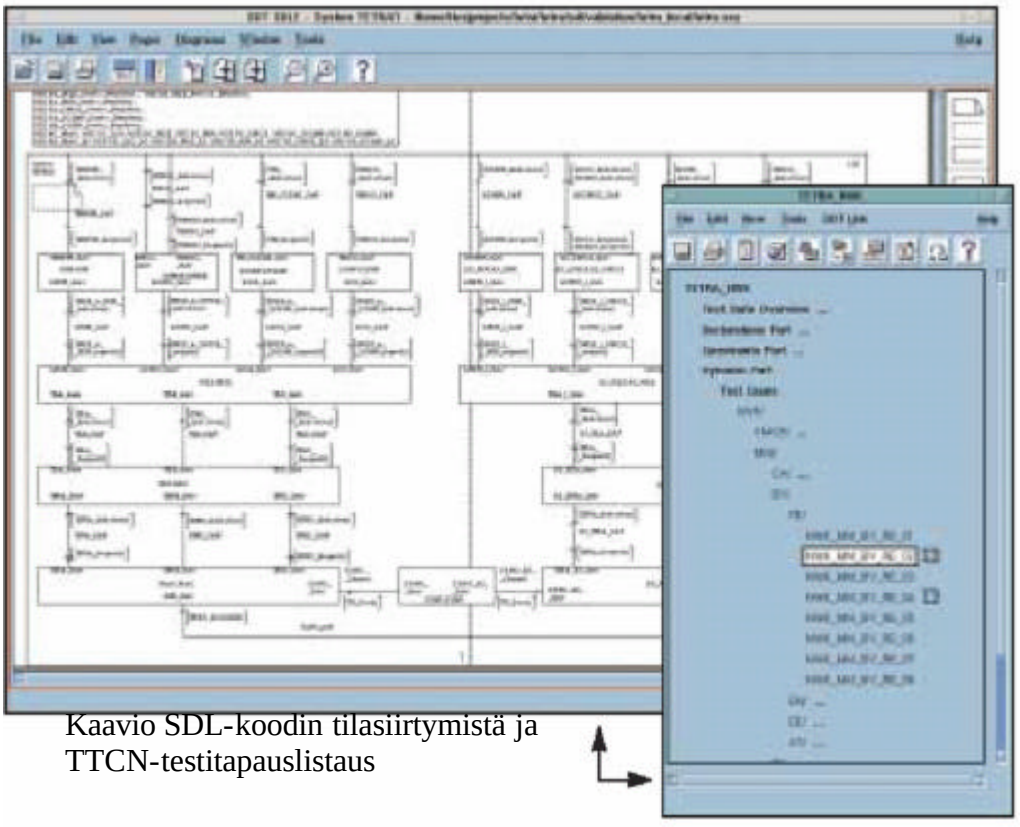
9.5.2. Sulautettujen järjestelmien testaaminen

On sanottu, että sulautettujen järjestelmien testaus alkaa siitä mihin tavallisen ohjelmiston testaus loppuu. Sulautetuissa järjestelmissä yhdistetään useita tekniikoita – elektroniikkaa, mekaniikkaa ja tietokonetekniikka. Jotta järjestelmä toimisi suunnitellulla tavalla, on eri tekniikoiden hallittu yhdistäminen tarpeen. Käytännössä integroinnin toimiminen arvioidaan resurssien ja aikataulujen rajoissa mahdollisimman kattavan testauksen avulla [Heikkinen *et al.*,. 1998].

Tyypillisesti järjestelmän mahdollisuudet hyödyntää ympäristön (esim. käyttöjärjestelmän) palveluita kehitysympäristössä eroavat olennaisesti kohdeympäristössä toimimisesta; sama toiminnallisuuden testaaminen joudutaan siis suorittamaan useampaan otteeseen Sulautettujen järjestelmien tyypillisesti epädeterministisestä luonteesta johtuen niiden testauksessa törmätään vielä verifioitavuusongelmiin; vaikka eilen saatiin tulos Y, ja tänään tulos X, molemmat toimintamallit voivat – tilanteesta riippuen - olla aivan yhtä oikeita. Sekä testitapausten automatisointi että tulosten automatisoitu käsittely voi siis olla vaikeaa, ja manuaalisen työn osuutta on vaikea pienentää.

Ongelmana sulautetuissa järjestelmissä on usein myös kohdeympäristön asettamat aika- ja muistirajat, ja vaikka testaaminen osittain korvattaisiinkin simuloinnilla, voi niidenkin verifiointi olla vaikeaa, esim. rajallinen muisti yhdistettynä hitaaseen tiedonsiirtonopeuteen voi aiheuttaa sen etteivät simulaatiosta saadut lokit ole täydellisiä, vaan välistä katoaa tapahtumatietoa.

Silloin kun kattava tilatestaus on mahdotonta tilojen suuren määrän vuoksi, saatetaan hyödyntää suoraa simulointia testauksen sijaan; näin voidaan käydä läpi vain testauksen kannalta oleelliset osat. Simuloimalla tehty testaus voidaan myös automatisoida erilaisten testikielien, kuten TTCN:n avulla [Heikkinen *et al.*,. 1998].



Kaavio SDL-koodin tilasiirtymistä ja TTCN-testitapauslistaus

Kuva 25. Työnäkymä SDL/TTCN –kehitysohjelmistosta.

9.5.3. Testauksen automatisoinnin edut ja hankaluudet

Testauksen automatisointi edellyttää haluttujen testitapausten tarkkaa dokumentaatiota, jotta voidaan arvioida, mitä testejä ylipäätään voi ja kannattaa yrittää automatisoida. Automatisointi ei siis synny helposti, vaan vie sekä aikaa että rahaa. Toimivinta automatisointi on siellä, missä tehtävä tarkastelu on rutiininomaista, esim. tietoturvallisuuden kannalta merkittävää hyötyä voidaan saavuttaa ohjelmiston tilasiirtymien automaattisella verifioinnilla: poikkeustilanteiden syntymistä voidaan näin ehkäistä tehokkaasti ja mahdollistaa nopea paluu normaalitilanteeseen. Toisaalta myös (esim. hyökkäyksen takia) pahasti vioittuneet ja siksi oudolla tavalla toimivat komponentit voidaan paremmin tunnistaa ja eristää muusta järjestelmästä.

Varsin usein automatisoinnin kuitenkin kuvitellaan tarkoittavan vain uuden testausympäristön hankintaa, käytön jäädessä sitten satunnaisten kokeilujen tasolle. Automatisoinnin harkittu keskittäminen juuri tietyille testauksen osa-alueille on kuitenkin tärkeää; testauksen automatisointia kokeilleista ohjelmistoprojekteista 26 % raportoi, ettei testaamisen automatisointi tuottanut mitään toivottuja tuloksia, ja jopa 2/3 projekteista taas ei saavuttanut minkäänlaisia säästöjä ajassa tai testausbudjetissa. Toisaalta testiympäristön testien toistettavuus ja siirrettävyys saattoi onnistui, jos siihen oli erityisesti pyritty [Pöyhönen, 2001].

Automatisointiin kiinnitetyt odotukset ovat usein epärealistisia, ja testaustoimintaan saatetaan edellytetään parannuksia joita ei ole edes kirjattu mihinkään testiympäristölle esitettyihin vaatimuksiin. Huono testauskäytäntö ei voi mitenkään parantua pelkällä automatisoinnilla; jos testausprosessi on epärealistisesti sijoitettu ohjelmistoprojektin elinkaaren, ajan puute voi edelleen estää kattavan testaamisen.

9.6. Automatisointi tulevaisuudessa

Automatisointiin perustuvat ohjelmiston kehitystyökalut myös mahdollistavat useiden eri ohjelmaversioiden tuottamisen samasta spesifikaatiosta (n.k. ”N-versio-ohjelmointi”) samassa ajassa. Tietoturvakriittisten järjestelmien tuottamisessa tämä voi olla merkittävä etu: ajamalla eri ohjelmistoversioita rinnakkain voidaan paremmin varmentaa systeemin tekemien ratkaisuiden oikeellisuus kaikissa tilanteissa (esim. äänestämällä) [Somerville, 2000], ja parantaa siten järjestelmän tietoturvatasoa. Silloin kun osa tarvituista komponenteista tuotetaan automatisoidusta, projektissa on myös mahdollista keskittää enemmän resursseja järjestelmän tietoturvan kannalta tärkeimpien osien kehitykseen, esim. jotakin automatisoitua koodin generointia varmempaa ohjelmistotuotantotapaa käyttäen.

Itse ohjelmointityönkin automatisoinnin houkuttelevuuden voi osaltaan katsoa johtuvan myös jatkuvasti lisääntyvästä testaustarpeesta; monimutkaistuvat systeemit ja lisääntyvä käytetyn ohjelmakoodin määrä altistavat sekä uusille hyökkäysuhkakuville että vaarallisille yhteensopimattomuusvirheille. Tällöin voi tuntua mielekkäältä pyrkiä säästämään esim. komponenttien yksikkötestauksessa, ja keskittää voimavarat koko järjestelmän täsmälliseen tietoturvatestaukseen. Etenkin vaikeasti testattavissa ympäristöissä, kuten sulautetuissa järjestelmissä, lisääntynyttä testaustarvetta voi ehkä helpottaa varhaisella simuloinnilla- ja mitä suuremmalta osalta tällaisten testitapausten käsittelyn voi automatisoida, sen enemmän resursseja jää järjestelmän kriittisten osien kehittämiseen, validointiin ja verifiointiin.

Toisaalta, myös automatisointityö pitää tarkastaa ja katselmoida; mikäli testitulokset käsitellään automaattisesti, testitapausten korkealaatuisiin on ensiarvoisen tärkeää. Etenkin tietoturvallisuuden näkökulmasta on erityisen vaarallista automatisoida jotakin sellaista, jonka toimintaperiaatteita ei tarkkaan ymmärretä. Testaamisen automatisointi ei myöskään auta testauksen laatuun, jos epämääräisyyden aiheutti epäselvä testitapausten dokumentointi ja testiajojen raportointi. On selvää, että koko testausprosessin parantamisella voi olla huomattavasti suuremmat vaikutukset projektin tuloksellisuuteen kuin pelkällä työkalupohjaisella automatisoinnilla.

Lähteet

[Ellsberger *et al.*,1997] Jan Ellsberger, Dieter Hogrefe, Amardeo Sarma, *SDL: Formal Object-Oriented Language for Communicating Systems.* Prentice Hall, 1997.

[Heikkinen *et al.*,. 1998] Jouni Heikkinen, Jukka Korhonen ja Tero Manninen, Sulautettujen ohjelmistojen testaus: Testausko vaikeaa? *Prosessori*, maaliskuu (1998), s. 93-95.

[Kangas, 2000] Jani Kangas, Automaattinen koodin generointi sulautetuissa järjestelmissä. *Insinöörityö*, Tampereen ammattikorkeakoulu, 2000.

[Kylmäkoski, 2002] Roope Kylmäkoski, Code generation from object oriented design. *Diplomityö*, Tampereen teknillinen yliopisto, 2002.

[Lisko, 2002] Hannu Lisko, CMICRO -koodingenerointi. *Insinäärityö*, Oulun ammattikorkeakoulu, 2002.

[Luukkainen, 2002] Matti Luukkainen, Formaalit menetelmät ja automaattinen verifiointi. *Esitelmä "Tietojenkäsittelytieteen esittely"*, Helsingin Yliopisto 7.2.2002.

[Mansikkamäki 2001] Reetta Mansikkamäki, Kuormitustestaus businessvaatimusten varmistaja. *Systeemityö*, 1 (2001), s. 2-4.

[Pöyhönen, 2001] Erkki Pöyhönen, Testauksen automatisointi edellyttää selkeitä tavoitteita. *Systeemityö*, 1 (2001), s. 8-9.

[Rumbaugh, 1997] James Rumbaugh, *OMT Insights : Perspective on Modeling from the Journal of Object-Oriented Programming*. Signature Sounds Recording, 1997

[Somerville, 2000] Ian Somerville, *Software Engineering (6th Edition)*. Addison-Wesley, 2000, 6. painos

[Telelogic, 2001] Telelogic, *Introduction to SDL*. Telelogic Academy, 2001.