# Luku3: Ohjelmien turvallisuus

Murtohyökkäykset käyttävät hyväkseen ohjelmistoissa olevia virheitä. Tässä luvussa katsotaan ensin tyypillisiä ohjelmiston virheitä, jotka antavat hyökkääjälle mahdollisuuden tunkeutua järjestelmään. Sen jälkeen annetaan muutamia ohjeita, joita noudattaen tietoturvahyökkäyksiä edesauttavat virheet vähenevät. Lisäksi tarkastellaan Javaa ja C:tä turvallisuuden näkökulmasta.

# Puskurin ylivuoto I

- Puskurin ylivuoto tapahtuu, kun datalle varattuun tiedostoon tai muistialueeseen yritetään viedä enemmän dataa kuin sinne mahtuu.
- Järjestelmästä riippuen ylimääräinen data voi valua muistipaikkoihin, joihin sen ei olisi tarkoitus tai joihin se ei saisi mennä. Jos hyökkääjä tietää, mikä osa datasta vuotaa yli, niin hyökkääjä voi saada omia ohjelmiaan järjestelmään.
- Puskurin ylivuoto voi johtua yksinkertaisesti siitä, ettei syötettä tarkisteta. Se voi myös johtua paljon vaikeammin havaittavasta ohjelmontivirheestä. Yleensä C-kieli on altis puskurin ylivuotohyökkäyksille, josta syystä esimerkit ovat C-kielellä.

Oletetaan, että ohjelmassa on määritelty 10 paikan taulukko:

```
char sample[10];
```

Jos ohjelmassa on koodi

```
for (i=0; i<=9; i++)
    sample[i]='A';
sample[10] ='B'
```

niin B vuotaa varatun muistialueen ulkopuolelle.

- Suorituksen aikana ohjelman koodi ja data on keskusmuistissa. Jos B menee ohjelman data-alueelle, se vaikuttaa jonkin muuttujan arvoon tai toistaiseksi tyhjään paikkaan.
- Jos taas se menee ohjelman koodialueelle ja korvaa sellaisen käskyn, joka on jo tehty ja johon ei enää palata, ei tapahdu mitään.
- Jos taas B menee koodialueelle muistipaikkaan, jossa on vielä suorittamaton käsky, se yritetään suorittaa jossain vaiheessa. Joka tapauksessa näissä tilanteissa vaikutukset rajoittuvat kyseiseen (sovellus)ohjelmaan.
- Seuraukset voivat olla vakavampia, jos taulukolle varatun alueen jälkeen alkaakin käyttöjärjestelmän oma data- tai koodialue.
- Jos esimerkiksi ylivuoto kohdistuu aliohjelmapinoon, hyökkääjä voi muuttaa paluuosoitetta haluamakseen, jolloin ohjelma siirtyy suorittamaan hyökkääjän koodia.

Edellä puskurin ylivuoto oli melko ilmeistä. Seuraava esimerkki näyttää, että ylivuotoa voi olla vaikea havaita.

**Esimerkki**. Tarkastellaan seuraavaa C-ohjelmaa:

```
int main(int argc, char **argv) {
  unsigned short int total;
  total = strlen(argv[1])+strlen(argv[2])+1;
  char *buff = (char *) malloc(total);
  strcpy(buf, argv[1]);
  strcat(buf, argv[2]);
}
```

Tyyppi `short int` on 16-bittinen. Valitaan `argv[1]` ja `argv[2]` siten, että `total` on 65537. Tyypistä johtuen `total` on itse asiassa 1. Puskurille `buff` varataan siten 1 muistipaikka ja puskurin ylivuoto seuraa. □

Puskurin ylivuotoihin joudutaan myös web-sovelluksissa. Esimerkiksi parametreja voidaan välittää http:n avulla:

```
http://www.somesite.com/subpage/userinput
&parm1=(09)8594535&parm2=2009Oct30
```

Nyt sivu `userinput` saa kaksi parametria. Asiakkaan selain hyväksyy arvot, koodaa ne ja välittää parametrit arvoineen palvelimelle. Jos sovellus on tehty huolimattomasti, asiakas voi aiheuttaa puskurin ylivuodon antamalla hyvin pitkät arvot parametreille.

# Tietojen epätäydellinen välitys (incomplete mediation)

- Tarkastellaan vielä edellistä http-esimerkkiä. Mitä tapahtuu, jos vuosi tai päivämäärä annetaan virheellisenä?
- Ehkä ohjelmassa nämä on otettu huomioon, ehkä ei. Yksi tapa ehkäistä syöttövirheitä on pakottaa käyttäjä valitsemaan ne ennalta annetusta listasta, jossa on vain laillisia arvoja.
- Jos kuitenkin valitut arvot välitetään palvelimelle URL:ssä, käyttäjä voi muuttaa URL:ää ennen kuin tiedot lähetetään palvelimelle. Palvelimella ei ole mitään mahdollisuutta ratkaista, tulivatko tiedot asiakkaan selaimelta suoraan vai muutettuna.

# Esimerkki: Parametrien välitys http-sanomassa I

- Seuraava esimerkki on todellisesta elämästä, vain nimiä on muutettu. Yhtiö Store myy tavaroita verkossa. Asiakas voi valita tavarat listasta.
- Ohjelma laskee oikein valittujen tavaroiden summan. Sen jälkeen ostaja voi valita, lähetetäänkö tavarat laivapostissa, tavallisessa postissa vai sähköisesti.
- Jos asiakas valitsi laivan, asiakkaan selain valmisti tilauksen:

  ```
  http://www.Store.com/order/final&custID=101&part=555A
  &qy=20&price=10&ship=boat&shipcost=5&total=205
  ```

- Siis asiakas 101 on ostanut 10:n arvoista tuotetta 555A 20 kpl. Lähetyskulut ovat 5 ja kokonaissumma 205.
- Pahantahtoinen asiakas saattoi nyt muuttaa kokonaissummaa haluamakseen URL-rivillä. Tällainen koodi oli jonkin aikaa tuotantokäytössä. Yhtiö kuitenkin käytti konsulttia tarkistamaan koodia, ja konsultti löysikin pian tämän aukon. □

# Synkronointivirhe I

- Tarkastellaan seuraavaa tilannetta. Käyttäjä pyytää pääsyä lukemaan tiedostoa antamalla tietueen, jossa on tiedoston nimi ja pääsyn tyyppi.
- Systeemin pääsynvalvontamekanismi tutkii tietueen. Se vertaa tiedoston nimeä, pääsyn tyyppiä ja käyttäjän identiteettiä talletettuihin pääsynvalvontatietoihin.
- Tämän mekanismi voi tehdä kopioimalla tietueen tiedot omiin puskureihinsa ja vertaamalla sitten puskureiden sisältöä pääsynvalvontarakenteissa oleviin tietoihin.
- Kun tarkistus on tehty, tietue välitetään eteenpäin sille systeemin osalle, joka välittää tiedot tiedostosta käyttäjälle.
- Näiden toimenpiteiden välissä käyttäjä voi muuttaakin tietueen sisältöä. Jos pääsynvalvontamekanismi hyväksyy pääsyn, niin välittäjälle annetaankin nyt eri tietue kuin mikä oli tarkistuksessa. Käyttäjä voi saada oikeudet hänelle kuulumattomiin tietoihin.

# SQL-solutus (SQL-injection) I

- Monilla sovelluksilla on rajapinta tietokantaan. Nämä rajapinnat saatttavat olla haavoittuvia,
  - jos tietokannan portti on piirikoodattu,
  - jos oletussalasananoja käytetään korkealla tasolla tai
  - jos käytetään merkkijonojen katenointia operaatioilla $+$, `concat()` tai `concatenate()`.
- Jälkimmäiset ovat vaarallisia erityisesti sovelluksissa, joissa SQL-kysely rakennetaan käyttäjän antamien parametritietojen pohjalta.
- Tällöin hyökkääjä voi syöttää pahantahtoista tietoa, jonka pohjalta sovellus rakentaa SQL-kyselyn, joka tekeekin muuta kuin mitä sovellus on tarkoittanut.

# SQL-esimerkki 1. I

Tarkastellaan SQL-lausetta

```
SELECT * FROM data WHERE id="+a-variable+";"
```

Tarkoituksena on, että muuttuja `a-variable` on luku, jonka perusteella taulukosta `data` valitaan tietue. Jos kuitenkin tarkistukset puuttuvat, ja asiakas antaakin merkkijonon, vaikkapa `1; DROP TABLE data`, niin lauseesta tulee

```
SELECT * FROM data WHERE id=1; DROP TABLE data
```

ja sen seurauksena taulu `data` poistetaan tietokannasta. Jotkut SQL-sovellusrajapinnat, kuten php:n mysql-query, eivät salli monia komentoja samalla kertaa. Siten näissä systeemeissä esimerkki ei onnistuisi. □

# SQL-esimerkki 2. I

Tarkastellaan lausetta

```
SELECT * FROM users where name='"+userName+";'"
```

Antamalla muuttujalle `userName` arvo `a' or 't'='t` lause muuttuu muotoon

```
SELECT * FROM users where name='a' or 't'='t'
```

Näin hyökkääjä pääsee käyttämään tietokantaa, vaikka hän ei tietäisi käyttöön oikeutettujen tunnuksia. □

Vuoden 2009 suuri luottokorttinumeroiden anastus Yhdysvalloissa (130 miljoona tapausta) perustui SQL-solutukseen.

# Virheet poikkeusten käsittelyssä I

- Ongelmia voi syntyä, jos koodissa ei reagoida riittävästi poikkeuksiin tai jos ei raportoida poikkeuksista.
- Poikkeusten hyödyntäminen hyökkäyksissä on ilmeisesti vähäistä, mutta niiden puutteellinen käsittely voi johtaa ohjelmiston virheelliseen toimintaan.
- Kuuluisin esimerkki lienee ranskalaisen Ariane 5 -avaruusraketin tuhoutuminen 1996 (lento 501) pääasiassa siitä syystä, ettei järjestelmä osannut käsitellä yhtä poikkeusta.
- Ohjelmisto tulkitsi virheellisesti poikkeuksen niin vakavaksi, että raketti oli tuhottava. Tosiasiassa poikkeuksen aiheuttanut virhetoiminta ei olisi vaatinut näin radikaalia toimenpidettä.

# Turvallisessa ohjelmoinnissa huomioon otettavia seikkoja

Tässä aliluvussa luetellaan useita tekijöitä, jotka täytyy ottaa huomioon turvallisessa ohjelmoinnissa. Nämä yleiset ohjeet eivät riipu ohjelmointikielestä. Ohjeet annetaan ohjelmointi- ja hallintosääntöinä. Katso tarkemmin Bishopin kirjasta Computer Security, chapter 29.5.

# 1. Prosessin oikeudet I

Yleensä prosessille tulisi antaa vain sen verran oikeuksia, kuin se minimissään tarvitsee. Kuitenkin prosessit voivat tarvita erilaisia oikeuksia eri suoritusvaiheissa.

**Implementointisääntö 1 (Moduulien eristämissääntö)**. Strukturoi prosessi (ohjelma) siten, että ylimääräisiä oikeuksia tarvitsevat prosessin osat ovat omana moduulinaan. Moduulien tulisi olla niin pieniä kuin mahdollista ja niiden tulisi tehdä vain niitä tehtäviä, joissa noita ylimääräisiä oikeuksia tarvitaan.

Tähän liittyy myös hallintosääntö:

**Hallintosääntö 1 (Oikeuksien hallintosääntö).** Tarkista, että oikeudet on asetettu asianmukaisesti.

# 2. Oikeudet pääsynvalvontatiedostoon I

Pääsynvalvontatietoja sisältävän tiedoston eheys on tärkeää. Eheys voidaan tarkistaa erityisillä ohjelmilla, mutta näitä ajetaan vain aika ajoin. Siten on tärkeää, että tarkistuksen tekee aina se ohjelma, joka tiedostoa käyttää.

**Hallintosääntö 2 (Prosessien suojelun hallintosääntö).** Prosesseja ja niihin liittyviä hallintatietoja luovaa ohjelmaa täytyy suojella niin, ettei sitä voi muuttaa ilman valtuutusta.

Monessa tapauksessa prosessi luottaa toisten tiedostojen tai ulkoisten resurssien asetuksiin. Aina kun se on mahdollista, ohjelman tulisi tarkistaa tällaiset riippuvuudet.

**Implementointisääntö 2 (Oletuksien tarkistussääntö).** Varmista, että ohjelma tarkistaa aina käyttäessään muita tiedostoja tai ulkoisia resursseja, että niihin liittyvät oletukset ovat voimassa. Jos tämä ei ole mahdollista, dokumentoi tällaiset tilanteet ylläpitäjiä varten, jotta he tietävät mahdolliset hyökkäyskohteet.

# 2. Oikeudet pääsynvalvontatiedostoon II

**Esimerkki.**

- Naivi tapa tarkistaa, että vain pääkäyttäjällä on oikeus kirjoittaa tiedostoon, on varmistua, että tiedoston omistaja on pääkäyttäjä ja tiedoston asetukset sallivat vain omistajan kirjoittaa siihen.
- Tarkastellaan ryhmäoikeuksia. Jos pääkäyttäjä on ryhmän ainoa jäsen, niin silloin ryhmän oikeudet sallivat ryhmän jäsenten kirjoittaa tiedostoon.
- Ongelmana on, että ryhmään kuulumisen tarkistaminen on mutkikkaampaa kuin ryhmän jäsenten selvittäminen. Käyttäjä voi kuulua ryhmään ilman, että hänet on listattu ryhmään kuuluvaksi, koska käyttäjän GID saadaan salasanatiedostosta, kun taas ryhmän jäsenyyslistat saadaan eri tiedostosta.
- Joko täytyy tarkistaa molemmat tiedostot tai ohjelman pitää ilmoittaa virheestä, jos joku toinen kuin käyttäjä voi kirjoittaa tiedostoon. □

# 3. Muistin suojaus I

**Implementointisääntö 3 (Muistinvarmistussääntö).** Varmista, ettei ohjelma jaa objekteja muistissa muiden ohjelmien kanssa ja että muut ohjelmat eivät pääse etuoikeutetun prosessin muistialueelle.

**Hallintosääntö 3 (Pienimmän oikeuden hallintosääntö).** Pakota muisti noudattamaan pienimmän oikeuden periaatetta.

Jos muistisektio ei tule sisältämään suoritettavia käskyjä, älä salli käskyjä noissa muistin osissa. Jos muistialuetta ei voi edes muuttaa, aseta se read-only -tyyppiseksi.

- Ohjelmassa nämä periaatteet voidaan ottaa huomioon kolmella eri tavalla.
- Ensinnäkin muuttumaton tieto voidaan tyypittää vakioiksi. Seuraa ajoaikainen virhe, jos suorituksen aikana yritetään muuttaa vakioita.
- Kaksi muuta tapaa liittyy tilanteeseen, jossa ohjelma ladataan muistiin.

# 3. Muistin suojaus II

- Lataaja vie tietoja kolmeen paikkaan: varsinaiselle data-alueelle, pinoon ja kekoon.
- Pinoa käytetään aliohjelmakutsujen yhteydessä ja kekoa dynaamisessa muistinvarauksessa.
- Näiden kolmen muistialueen muistipaikat tulisi tehdä sellaisiksi, ettei niitä voi suorittaa (execute).
- Data- ja pinoalueen kohdalla tämä ei aiheuta ongelmia. Itse asiassa uusimmat versiot käyttöjärjestelmistä asettavat nämä alueet oletusarvoisesti non-executable -tilaan.
- Jos sen sijaan kekoa ei saa käyttää suoritukseen, dynaaminen lataus ei onnistu. Ts. funktioiden ajoaikainen lataaminen ei onnistu. On usein mahdollista kääntää ohjelma siten, ettei se käytä dynaamista latausta.

# 4. Luottamus

Yksittäinen ohjelma käyttää hyväkseen monia systeemin komponentteja. Esimerkiksi systeemi voi luottaa systeemin todentamismenetelmiin käyttäjän tunnistamiseksi tai se voi luottaa käyttäjätietoja ylläpitävään tietokantaan asettaakseen käyttäjien roolit.

**Hallintosääntö 4 (Komponenttien tunnistamisen hallintosääntö).** Tunnista kaikki systeemin komponentit, joita ohjelmasi joutuu käyttämään. Tarkista komponenttien virheettömyys aina kun mahdollista ja identifioi ne komponentit, joiden kohdalla virhetarkistuksia ei voi tehdä.

# 5. Implementaation yksityiskohtien vajavainen eristäminen

Tämä tarkoittaa montaa asiaa. Yleissääntö on seuraava.

**Implementointisääntö 4 (Virhetilanteiden sääntö).** Jokaisen funktion käyttäytyminen virhetilanteissa täytyy selvittää. Älä yritä toipua, jos ei ole varmaa, etteivät seuraukset aiheuta turvallisuusongelmia. Ohjelman tulisi palauttaa systeemin tila samaksi kuin ennen alkua ja sitten lopettaa.

Tarkastellaan erilaisia tilanteita, joissa em. säännöllä on merkitystä.

# 5.1 Resurssien kuluttaminen ja käyttäjien tunnukset I

- Tarkastellaan esimerkkiä, jossa postipalvelin salli käyttäjien edelleen lähettää (forward) viestejä luomalla uuden tiedoston FF (forwarding file).
- FF määritteli tiedostot, joihin edelleen jaettavat viestit viedään. Tässä tapauksessa postipalvelin välitti viestin ja liitti viestiin FF:n omistajan oikeudet (uid).
- Joissakin tapauksissa postipalvelin vei viestejä jonoon lähettääkseen niitä myöhemmin. Kun se teki niin, se kirjoitti käyttäjän nimen (ei uid:tä) kontrollitiedostoon.
- Nimi saatiin tietokannasta käyttäen uid:tä avaimena. Jos kysely epäonnistui, postipalvelin käytti ylläpitäjien asettamaa oletusnimeä.
- Hyökkääjät huomasivat, kuinka kysely saadaan epäonnistumaan. Sen seurauksena käyttäjäksi tuli oletuskäyttäjä, yleensä systeemitason olio (demoni).

# 5.1 Resurssien kuluttaminen ja käyttäjien tunnukset II

- Tämä teki hyökkääjille mahdolliseksi lisätä tietoa mihin tahansa tiedostoon, johon oletuskäyttäjällä oli oikeudet.
- Hyökkääjät käyttivät tätä mahdollisuutta hyväkseen asentaessaan troijalaisia systeemiohjelmiin. Näin he saivat ylimääräisiä oikeuksia ja systeemi oli haavoittunut.
- Lopulta tällaiset hyökkäykset torjuttiin muuttamalla ohjelmaa niin, että se pysähtyi, jos nimikysely epäonnistui.

# 5.2. Pääsynvalvontatietojen validointi I

- Pääsynvalvontatiedot määrittelevät pääsynvalvontapolitiikan. Yleensä politiikka määritellään ensin, ja se on melko abstraktilla tasolla.
- Oleellinen kysymys onkin, vastaako pääsynvalvontatieto politiikkaa. Tämä tulisi tarkistaa erityisesti muutosten jälkeen.
- Eräässä tapauksessa ohjelmoijat kehittivät toisen ohjelman, joka käytti samoja rutiineja kuin alkuperäinen, rooleihin perustuva ohjelma ja joka analysoi pääsynvalvontatietoja.
- Tämä uusi ohjelma tulosti pääsynvalvontatietoja helposti luettavassa muodossa. Se mahdollisti, että ylläpitäjät tarkistivat pääsynvalvontatietojen oikeellisuuden. Tarkistus tehtiin aika ajoin ja aina muutosten jälkeen.

# 5.3 Roolin toteuttavan prosessin oikeuksien rajoittaminen I

- Roolin toteuttava prosessi voidaan luoda kahdella tavalla.
- UNIX:ssa ohjelma voi luoda lapsiprosessin (fork-rutiini).
- Se voi myös aloittaa uuden ohjelman siten, että aloittaja korvataan uudella prosessilla. Tämä tekniikka, jota englanniksi kutsutaan toisinaan termillä “overlaying” ja joka UNIX:ssa toteutetaan exec-käskyllä, on tässä tilanteessa yksinkertaisempi kuin lapsiprosessin luominen.
- Se mahdollistaa, että uuden prosessin oikeudet ovat vähäisemmät kuin alkuperäisen, jos rooli niin vaatii.
- Tarkastellaan tilannetta, jos käytetään tätä menetelmää. Uusi prosessi perii alkuperäisen oikeudet.
- Ennen uuden prosessin käynnistämistä alkuperäisen prosessin täytyy asettaa oikeutensa roolia vastaavaksi.

# 5.3 Roolin toteuttavan prosessin oikeuksien rajoittaminen II

- Tämä voidaan tehdä siten, että suljetaan kaikki tiedostot, jotka alkuperäinen prosessi oli avannut ja muuttamalla prosessin oikeudet roolia vastaavaksi.
- Erityistä huomiota täytyy kiinnittää UID:ien ja GID:ien käsittelyyn. Efektiiviset UID:t ja GID:t kontrolloivat oikeuksia. Sen tähden ohjelmoijat palauttivat efektiivisen GID:n ensiksi oletusarvoihin ja sen jälkeen UID:n. (Jos palautus olisi tehty toisinpäin, GID:ien muutokset epäonnistuisivat, koska näiden muutosten tekeminen vaatisi juurioikeuksia.)
- Jos kuitenkin UNIX-systeemi tukee talletettuja UID-arvoja, valtuutettu käyttäjä saattaa kyetä hankkimaan juurioikeudet, vaikkei roolilla niitä olekaan.
- Ongelmana on, että efektiivisen UID-arvon asetus asettaa talletetuksi UID-arvoksi edellisen UID-arvon eli juuren. Prosessi voi sitten palauttaa tämän UID-arvon.

# 5.3 Roolin toteuttavan prosessin oikeuksien rajoittaminen III

- Välttääkseen tämän ohjelmoijat käyttivät setuid-systeemikomentoa asettaakseen kaikki todelliset, efektiiviset ja talletetut UID-arvot roolin mukaiseksi.

# 6. Vajavainen muutos

Muutetut data-arvot ja käskyt voivat olla ristiriidassa aikaisempien arvojen kanssa, jos muutoksia ei ole tehty hallitusti. Seurauksena voi olla suoritus, joka on virheellinen tai turvaton. Tarkastellaan muutoksia keskusmuistissa ja tiedostoissa sekä pääsynhallintaa tiedostoihin.

# 6.1 Keskusmuisti I

Tietoa voi olla jaetussa muistissa. Jokainen jaettuun muistiin oikeutettu prosessi voi muuttaa jaetun muistin tietoja. Tällaisissa tilanteissa rinnakkaisuuden hallinta tulee oleelliseksi.

**Esimerkki.**

- Kaksi prosessia jakaa muistialueen. Toinen prosessi lukee todentamiseen liittyvää tietoa ja kirjoittaa sen jaettuun muistiin.
- Toinen prosessi suorittaa todennuksen ja kirjoittaa Boolen arvon `tosi` jaettuun muistiin, jos todennus onnistuu.
- Muussa tapauksessa prosessi kirjoittaa `epätosi`.
- Jos rinnakkaisuuden hallinnasta ei välitetä, ensimmäinen prosessi voi lukea tuloksen ennenkuin toinen on ehtinyt kirjoittaa sen. Tuloksena voi olla pääsy kiellettyihin tietoihin tai pääsyn kielto, vaikkei siihen olisi perusteita. □

**Implementointisääntö 5 (Vuorovaikutuksein synkronointisääntö).** Jos prosessilla on vuorovaikutusta muiden prosessien kanssa, vuorovaikutukset tulee synkronoida. Erityisesti tulee tuntea kaikki erilaiset suorituspolut, joita vuorovaikutuksissa voi syntyä. Kaikkien suorituspolkujen tulee noudattaa valittua turvapolitiikkaa.

Samankaltainen tilanne voi syntyä asynkronisen poikkeusten käsittelijän yhteydessä. Jos käsittelijä muuttaa muuttujien arvoa ja palauttaa kontrollin poikkeusta edeltävään kohtaan ohjelmassa, muutokset saattavat aiheuttaa ongelmia. Siitä syystä ohjelmoijan tulee tuntea muutosten seuraukset, jotka liittyvät muuttujiin, joista muut ohjelman osat ovat riippuvaisia.

**Implementointisääntö 6 (Poikkeuskäsittelijöiden sääntö).** Asynkronisten poikkeuskäsittelijöiden ei tulisi muuttaa mitään muita muuttujia kuin poikkeuskäsittelijän sen hetkisen moduulin lokaaleja muuttujia. Poikkeuskäsittelijän tulisi estää muut poikkeukset sen jälkeen

# 6.1 Keskusmuisti III

kun poikkeus on tapahtunut, eikä estoa tulisi purkaa ennenkuin ko. poikkeuksen käsittely on päättynyt. Poikkeuksia tästä säännöstä voidaan tehdä vain, jos suunnittelussa otetaan huomioon poikkeukset poikkeusten sisällä.

- Toinen mahdollisuus väärinkäyttöön syntyy, kun käyttäjä syöttää ohjelmalle virheellisen tyyppistä tietoa.
- Tämä voi johtaa puskurin ylivuotoon, joka muuttaa pinossa olevaa paluuosoitetta. Tällaiset muutokset on mahdollista havaita.
- Heti kun paluuosoite on viety pinoon, viedään sinne myös satunnnaisluku (engl. canary). Jos syöte vuotaa muistialueensa yli ja muuttaa paluuosoitetta, se todennäköisesti muuttaa myös satunnaislukua.
- Kun paluuosoite haetaan pinosta, palautetaan ensin satunnaisluku ja verrataan sitä vietyyn lukuun. Jos nämä kaksi eroavat, ylivuoto on tapahtunut. Tietenkin tällainen tarkistus täytyy toteuttaa kääntäjässä.

# 6.1 Keskusmuisti IV

Toisenlainen ylivuototilanne syntyy seuravassa esimerkissä.

- Eräässä UNIX-versiossa login-ohjelmassa käytettiin kahta peräkkäistä taulukkoa.
- Ensimmäiseen, 80 merkkiä pitkään taulukkoon oli tarkoitus viedä käyttäjän antama selväkielinen salasana.
- Toiseen, 13 merkkiä pitkään taulukkoon oli viedä salasanasta laskettu tiivistearvo, joka saatiin salasanatiedostosta.
- Kun käyttäjä oli antanut nimensä, ohjelma haki heti nimeä vastaavan salasanan tiivisteen lyhyempään taulukkoon.
- Sen jälkeen ohjelma luki käyttäjän antaman salasanan ja vei sen pitempään taulukkoon. Sitten ohjelma laski pitemmässä taulukossa olevan sanan tiivisteen ja vertasi sitä toisen taulukon tiivisteeseen. Jos nämä olivat samoja, käyttäjä oli todennettu.
- Hyökkääjä kirjautui sisään juurena. Hän valitsi mielivaltaisen salasanan, jolle oli laskenut etukäteen tiivisteen.

# 6.1 Keskusmuisti V

- Kysyttäessä salasanaa hän antoi valitsemansa salasanan, kirjoitti sen jälkeen tyhjää niin, että 80 merkkiä tuli täyteen ja vielä lopuksi tiivisteen 13 merkkiä. Syöte vuoti yli pitemmästä taulukosta ja lyhyempään tulikin tekaistusta salasanasta laskettu tiiviste. Hyökkääjä todennettiin.

**Implementointisääntö 7 (Luotettavan datan sijoitussääntö).** Aina kun mahdollista, pidä prosessin luotettava data eri paikassa kuin data, joka tulee epäluotettavalta taholta. Huolehdi myös, että muistivirhe syntyy, jos epäluotettava data korvaa luotettavan.

Käytännön ohjelmointitoimia säännön toteuttamiseksi ovat: ei käytetä muuttujia uudelleen, jos niihin on luettu syötettyä dataa ja tarkistetaan syötteet ohjelman sisällä.

# 6.2 Tiedostojen sisällön muutokset I

- Tiedoston sisältö voi muuttua virheellisesti. Useimmissa tapauksissa tämän syynä ovat tiedoston virheelliset pääsyoikeudet tai rinnakkain toimivien prosessien tiedostoviittaukset. Hallintosääntö 2 ja implementointisääntö 5 huolehtivat näistä tilanteista.
- On myös oltava huolellinen, jos ohjelmassa tapahtuu dynaamisia latauksia. Dynaamiset kirjastot eivät ole osa ohjelman suoritettavaa koodia. Ne ladataan tarvittaessa ohjelman ajon aikana.
- Oletetaan, että yksi dynaaminen moduuli on muuttunut ja että se aiheuttaa sivuvaikutuksen. Ohjelma voi keskeytyä tai jopa toimia sen johdosta virheellisesti.
- Jos dynaamisia moduuleita ei voi muuttaa, riskiä ei silloin juuri ole. Mutta jos moduuleita voidaan päivittää tai muuten muuttaa, tilanne on vakavampi.

# 6.2 Tiedostojen sisällön muutokset II

- Dynaamisten moduulien yksi päätarkoitushan on välttää koko ohjelman kääntäminen päivityksen yhteydessä, joten turvallisuuteen liittyvät, dynaamisia kirjastoja käyttävät ohjelmat ovat vaarassa.

**Implementointisääntö 8 (Epätoivottavien komponenttien sääntö) .**
Älä käytä komponentteja, jotka voivat muuttua ohjelman ajojen välillä.

# 6.3 Kilpatilanteet tiedostojen käsittelyssä I

- Nämä tilanteet tarkoittavat pääsyehtojen tarkistuksen ja tiedoston käytön välistä aikaa, jolloin asiaankuulumattomia muutoksia voi tapahtua.
- Jotta tällaisia muutoksia ei tapahtuisi, joko tiedosto täytyy suojata niin, etteivät asiaankuulumattomat pääse operoimaan sitä, tai prosessin täytyy validoida tiedosto ja käyttää sitä jakamattomasti.
- Edellinen vaatii sopivat pääsyasetukset, joten siihen sopii hallintosääntö 2. Jälkimmäisestä puhutaan enemmän kohdassa *Virheellinen jakamattomuus*.
- Tarkastellaan esimerkkinä UNIXin xterm-ohjelmaa. Se emuloi päätettä x11-ikkunaympäristössä. Syistä, jotka eivät ole oleellisia tässä esimerkissä, sitä täytyy ajaa juurioikeuksin.
- Ohjelma vie käyttäjän kaiken syöte- ja tulostustiedon log-tiedostoon. Jos tiedosto on jo olemassa, xterm tarkistaa, että käyttäjä voi kirjoittaa sinne ennen kuin tiedosto avataan.

# 6.3 Kilpatilanteet tiedostojen käsittelyssä II

- Koska juurikäyttäjä voi kirjoittaa mihin tahansa tiedostoon, ylimääräinen tarkistus on tarpeen, jotta estettäisiin käyttäjää ohjaamasta xtermin avulla log-tiedoston sisältöä esimerkiksi salasanatiedostoon ja tällä muuttamasta salasanatiedostoa.
- Oletetaan, että käyttäjä haluaa kirjoittaa olemassaolevaan tiedostoon. Seuraava koodi avaa tiedoston kirjoittamista varten:

```
if (access("/usr/tom/X", W_ok)==0) {
    if ((fd=open("/usr/tom/X", O_WRONLY|O_APPEND))<0 {
        /* handle error: cannot open file */
    }
}
```

- UNIXissa tiedoston nimi on löyhästi sidottu vastaavaan dataobjektiin. Sidos varmistetaan joka kerran kun nimeä käytetään.

# 6.3 Kilpatilanteet tiedostojen käsittelyssä III

- Jos nimeä `/usr/tom/X` vastaava tiedosto vaihdetaan access- ja open-operaation välissä, open ei avaa samaa tiedostoa, minkä access on varmentanut.
- Jos siis tässä välissä hyökkääjä poistaa tiedoston ja linkittää tilalle systeemitiedoston (alias, symboliset linkit), esimerkiksi salasanatiedoston, hyökkääjä voi luoda juuriprosessin ilman salasanaa ja saada näin juurioikeudet.

# 7. Virheellinen nimentä I

- Otsake viittaa tilanteeseen, jossa objekti on nimetty epäselvästi. Tavallisimmin kahdella objektilla on sama nimi.
- Kuvitellaan, että ohjelmoija aikoo käyttää nimeä toiseen objektiin, mutta hyökkääjä manipuloi ympäristöä siten, että nimi viittaakin toiseen. Jotta tämän kaltaisilta tilanteilta vältyttäisiin, objektit pitää nimetä yksikäsitteisesti. Tämä liittyy sekä hallinnointiin että implementointiin.
- Objektin tulee olla identifioitavissa yksikäsitteisesti tai täydellisesti vaihdettavissa. Näiden objektien hallinnoimisella tarkoitetaan, että identifioidaan ne, jotka ovat vaihdettavissa, ja ne, jotka eivät ole.
- Edelliset tarvitsevat kontrollimoduulin, joka nimen saadessaan valitsee yhden vaihtoehtoisista objekteista. Jälkimmäiset taas vaativat yksikäsitteisen nimen.

# 7. Virheellinen nimentä II

**Hallintosääntö 5 (Nimien hallintosääntö).** Yksikäsitteiset objektit tarvitsevat yksikäsitteisen nimen. Vaihdettavat objektit voivat jakaa nimen.

Nimi tulkitaan tietyssä kontekstissa eli yhteydessä. Implementointitasolla prosessin täytyy pakottaa oma kontekstinsa sovitun tulkinnan mukaiseksi. Konteksti sisältää tietoa merkistöstä, prosessi- ja tiedostohierarkiasta, verkkoympäristöstä ja käytettävistä muuttujista kuten hakupoluista.

**Implementointisääntö 9 (Kontekstisääntö).** Prosessin täytyy varmistaa, että konteksti, jossa objekti nimetään, määrittelee oikean objektin.

- Tarkastellaan hypoteettista ohjelmaa, joka käyttää neljää luokkaa nimiä: pääsynvalvontatiedoston nimeä, käyttäjien ja roolien nimiä, koneiden nimiä ja komentotulkin nimeä, jota ohjelma käyttää suorittaessaan roolien käskyjä.

# 7. Virheellinen nimentä III

- Pääsynvalvontatiedoston ja komentotulkin nimen täytyy identifioida tiedosto yksikäsitteisesti. Absoluuttiset polkunimet määrittelevät objektin sijainnin juurihakemiston suhteen. Etuoikeutettu prosessi voi määritellä minkä tahansa hakemiston juurihakemistoksi, mutta tarkasteltava ohjelma ei sitä tee. Siten absoluuttiset polkunimet määrittelevät tiedostot yksikäsitteisesti.
- Annettu nimi voidaan tulkita eri tavalla eri ympäristöissä. Esimerkiksi erot merkkien koodauksessa voivat johtaa siihen, että nimi muuttuu. Myös jotkut ympäristömuuttujat voivat aiheuttaa nimimuutoksia.
- Tarkastelteltava ohjelma luo oman, luotettavan ympäristön käskyjen toteuttamiseksi. Tällä on kaksi etua verrattuna siihen, että ohjelma tutkisi tarkasti ympäristön.
- Ensinnäkin vältetään tämä operaatio, joka saattaa sisältää monia yksityiskohtia.

# 7. Virheellinen nimentä IV

- Toiseksi, tämä ratkaisu mahdollistaa sen, että systeemiä voidaan kehittää vaarantamatta turvallisuutta. Esimerkiksi jos uutta ympäristömuuttujaa käytetään ohjaamaan ohjelman toimintaa, muuttuja ei vaikuta ohjelman käskyjen suoritukseen, koska muuttuja ei ole käskyn ympäristössä.
- Tarkasteltava ohjelma siis sulkee kaikki tiedostokuvaajat, palauttaa signaalinkäsittelijät alkutilaansa ja vie uuden ympäristömuuttujajoukon käskylle. Nämä toimenpiteet toteuttavat implementointisäännön 9.
- Ohjelman kehittäjät olettivat, että systeemiä ylläpidetään oikein siten, että käyttäjiä ja rooleja vastaavat UID:t ovat oikein. Tätä vastaa hallintosääntö 5.
- Koneiden ja verkkolaitteiden nimet muodostavat neljännen nimijoukon. Nämä voidaan identifioida joko nimen tai IP-osoitteen avulla.

# 7. Virheellinen nimentä V

- Jos käytetään nimiä, niiden tulee olla täydellisiä DN-nimiä, jotta vältyttäisiin monimielisyyksiltä. Jos esimerkiksi pääsynvalvonta sallii Matille roolin myyjä, kun tämä kirjautuu sisään amelia-koneelta, niin tarkoittaako tämä, että amelia on paikallisen verkon kone vai minkä tahansa verkon saman niminen kone? Kumpikin tulkinta voisi olla oikea, mutta tässä tapauksessa oletetaan ensimmäinen vaihtoehto. Tämä ratkaisu toteuttaa implementointisäännön 9.
- Jos paikallinen verkko on väärin konfiguroitu tai sitä ylläpidetään huonosti, nimi amelia voikin viitata muuhun kuin tarkoitettuun koneeseen. Esimerkiksi todellinen amelia voi kaatua. Hyökkääjä voi sen jälkeen asettaa oman koneensa vastaamaan ameliaa. Ohjelma ei ota huomioon tällaisia mahdollisuuksia.

# 8. Virheellinen resurssien vapauttaminen tai tiedostojen poistaminen I

Jos arkaluonteisen tiedon poistaminen tehdään huolimattomasti, toinen käyttäjä tai prosessi voi nähdä tietoa jälkeenpäin. Erityisesti kryptografiset avaimet, salasanat ja muu todentamiseen ja luottamuksellisuuteen liittyvä tieto tulisi tuhota käytön jälkeen. Samalla tavalla resurssi tulisi vapauttaa, kun sitä ei enää tarvita. Jos vapauttaminen tehdään huolimattomasti tai virheellisesti, toiset prosessit voivat käyttää resurssia esimerkiksi palvelunestohyökkäykseen.

**Implementointisääntö 10 (Luottamuksellisten objektien tuhoamissääntö)**. Kun prosessi ei enää käytä luottamuksellista objektia, objekti tulisi puhdistaa ja sen jälkeen tuhota tai vapauttaa.

# 8. Virheellinen resurssien vapauttaminen tai tiedostojen poistaminen II

- Tarkastellaan taas hypoteettista ohjelmaa, joka käyttää kolmen tyyppistä sensitiivistä tietoa. Ensimmäinen on selväkielinen salasana, joka todentaa käyttäjän. Salasanasta lasketaan tiiviste, jota verrrataan talletettuun tiivisteeseen. Kun tiiviste on laskettu, prosessi tuhoaa selväkielisen salasanan kirjoittamalla salasanan sisältävään taulukkoon uutta, satunnaista bittijonoa.
- Toisen ryhmän sensitiivistä tietoa muodostavat pääsynvalvontatiedot. Oletetaan, että hyökkääjä haluaa päästä käsiksi jonkun roolin tiedostoihin. Pääsynvalvontatiedot kertoisivat, keillä on pääsy kyseisiin tiedostoihin.

# 8. Virheellinen resurssien vapauttaminen tai tiedostojen poistaminen III

- Estääkseen hyökkääjää pääsemästä käsiksi näihin tietoihin ohjelmoijat päättivät pitää ne luottamuksellisena. Ohjelma lukee näitä tietoja käyttäen tiedostokuvaajia. Oletusarvoisesti tiedostokuvaajat jäävät auki, kun ohjelma käynnistää muita prosesseja. Tästä syystä ohjelma sulkee pääsynvalvontatiedoston tiedostokuvaajan, kun todennuspyyntö on käsitelty.
- Kolmas ryhmä sensitiivistä dataa on lokitiedosto. Ohjelma muuttaa tätä tiedostoa. Jos tavallinen prosessi perisi tiedostokuvaajan, se voisi täyttää lokitiedoston, jolloin ohjelmamme ei voisi enää kirjata epäonnistumisia. Siksi ohjelma sulkee myös lokitiedoston ennen kuin ryhtyy toteuttamaan roolin käskyä.

# 9. Vaillinainen validointi

Ongelmia syntyy, jos tiedon oikeellisuutta ja konsistenssia ei tarkisteta. Seuraavassa on muutamia tilanteita, joissa tieto tulisi validoida. Lista ei ole kattava, vaan jokaisen ohjelman kohdalla pitää miettiä myös muita tilanteita.

# 9.1 Rajojen tarkistus I

Jos esimerkiksi taulukko on määritelty välille $0, \ldots, 99$, on tärkeää, että indeksi on juuri tällä välillä.

**Implementointisääntö 11 (Taulukkoviittausten sääntö).** Varmista, että kaikki taulukkoviittaukset kohdistuvat todella taulukkoon. Jos jonkun funktion kohdalla ei voi olla varma viittausrajoista, älä käytä kyseistä funktiota. Etsi toinen funktio, kirjoita uusi versio tai luo kääre (wrapper).

Säännön mukaiset tarkistukset on helppo tehdä omassa ohjelmassa, mutta kirjasto-ohjelmien kohdalla tilanne on monimutkaisempi.

**Esimerkki.** Tarkastellaan merkkijonojen kopiointia.

- C:ssä on käytössä sopimus, että merkkijono esitetään merkkitaulukkona, jonka lopussa on 0.
- Jonojen kopiointiin on olemassa funktio `strcpy(x,y)`, joka kopioi taulukossa $y$ olevan merkkijonon taulukkoon $x$.

# 9.1 Rajojen tarkistus II

- Tämä tapahtuu, vaikka $y$ olisi pitempi kuin $x$.
- Toinen funktio, `strncpy(x,y,n)` kopioi korkeintaan $n$ merkkiä $y$:stä $x$:ään. Se voi kuitenkin jättää loppumerkin 0 kopioimatta.
- Jos nyt käytetään tätä funktiota, niin turvallisuuden vuoksi 0 täytyy erikseen sijoittaa $x$:n loppuun. Lisäksi prosessin tulisi tarkistaa, että $x$ ja $y$ ovat todella merkkijonotaulukoita ja että $n$ on sopiva kokonaisluku. □

# 9.2 Tyyppitarkistus I

**Implementointisääntö 12 (Tyyppien tarkistussääntö).** Tarkista funktioiden ja parametrien tyypit

**Hallintosääntö 6 (Kääntämisen hallintosääntö).** Kun ohjelmia käännetään, varmista, että kääntäjä ilmoittaa kaikista tyyppien ristiriitaisuuksista. Tutki kaikki sellaiset ilmoitukset ja korjaa probleema tai dokumentoi se ja sen vaarattomuus

# 9.3 Virhetarkistukset I

Yleisin virhe on jättää funktioiden paluuarvot huomiotta. Tarkastellaan esimerkiksi tilannetta, jossa ohjelman täytyy selvittää tiedoston omistaja. Ohjelma käyttää systeemifunktiota, joka palauttaa omistajatietoja sisältävän tietueen. Jos funktio epäonnistuu, tietueen tiedot ovat arvottomia.

**Implementointisääntö 13 (Proseduurien tulosten seurantasääntö).** Tarkista kaikkien funktio- ja proseduurikutsujen yhteydessä, ettei kutsujen suorituksessa ole tapahtunut virheitä.

# 9.4 Tiedon oikeellisuuden, ei virheellisyyden tarkistus I

Kaikki validit arvot tulisi tuntea ja muut hylätä. Valitettavasti ohjelmoijat usein tarkistavat vain virheellisen tiedon ja olettavat, että loput ovat aina oikein.

**Esimerkki.**

- Metamerkki on merkki, joka tulkitaan muuksi kuin mitä se varsinaisesti on. Esimerkiksi merkki ? on UNIXin komentotulkille metamerkki, joka esittää mitä tahansa merkkiä.
- Eräässä uudessa, päivitetyssä komentotulkin versiossa merkki ‘ tulkittiin käskyn erottimeksi eli metamerkiksi. Vanha versio tulkitsi merkin tavalliseksi merkiksi.
- Komentotulkissa oli myös osa, joka suoritti käskyjä verkon yli. Sallittujen käskyjen joukko oli rajoitettu. Ohjelma tutki huolellisesti, että käsky oli sallittu ja ettei se sisältänyt metamerkkejä ennenkuin käsky lähetetiin kaukaiselle komentotulkille.

# 9.4 Tiedon oikeellisuuden, ei virheellisyyden tarkistus II

- Valitettavasti ohjelma tarkisti vain ne metamerkit, jotka piti hylätä, ei niitä, jotka sallittiin käskyissä. Seurauksena oli, että hyökkääjä saattoi ujuttaa kielletyn käskyn sallittujen joukkoon, koska metamerkkien listaa ei oltu päivitetty merkille '. □

**Implementointisääntö 14 (Sallittujen arvojen sääntö).** Tarkista, että muuttujan arvot ovat sallittuja.

**Hallintosääntö 7 (Vaarallisten piirteiden hallintosääntö).** Jos on tarpeellista ottaa huomioon muita, turvallisuutta vaarantavia piirteitä, dokumentoi perustelut, mahdolliset vaikutukset ja tilanteet, joissa piirteitä pitää käyttää. Näin muut voivat arvioida riskejä. jotka syntyvät piirteiden käyttämisestä.

**Implementointisääntö 15 (Syötteiden tarkistussääntö).** Tarkista kaikki syöte sekä muodon että sisällön osalta. Erityisesti tarkista kokonaisluvuista, etteivät ne ole liian isoja tai pieniä, ja tarkista merkkitieto, että se on sopivan pituinen eikä sisällä muuta kuin sallittuja merkkejä.

Esimerkiksi annettu salasana ei saa olla pitempi kuin mitä puskuriin mahtuu. Tilanteet voivat olla myös monimutkaisempia kuten seuraava esimerkki näyttää.

**Esimerkki.** C:n printf-funktion ensimmäinen parametri on merkkijono, joka ilmaisee, miten tulostus muotoillaan. Seuraavat parametrit sisältävät datan. Siten

```
printf("%d%d\n",i,j)
```

tulostaa arvot $i$ ja $j$. Jotkut versiot sallivat, että tulostuskäskyn yhteydessä talletetaan tulostettujen merkkien lukumäärä. Jos $i = 2$ ja $j = 21$, niin

```
printf("%d%d%m%d\n%n", i,j,&m,i,&n)
```

tulostaa

```
2, 21 2
```

ja tallettaa 4:n *m*:ään ja 7:n *n*:ään, koska ennen %m:ää on tulostettu 4 merkkiä ja 7 merkkiä ennen %n:ää (\n tulkitaan yhdeksi merkiksi, rivinvaihdoksi). Oletetaan, että käyttäjältä kysytään tiedostonimeä. Tämä nimi viedään taulukkoon *str*. Sen jälkeen ohjelma tulostaa `printf(str)`. Jos käyttäjä antaa tiedostonimen `log%n`, tulostus kirjoittaa johonkin muistipaikkaan luvun 3. Täsmällinen paikka riippuu ohjelmapinosta ja kokeilemalla saattaa onnistua, että käsky muuttaa paluuosoitetta. Tämä johtaa puskurin ylivuotohyökkäykseen. □

# 9.6 Validoitavat ratkaisut

Toisinaan dataa ei voi validoida täydellisesti. Esimerkiksi C:ssä on mahdollista testata, onko osoitin NULL vai ei, mutta on vaikeaa ellei mahdotonta testata, osoittaako osoitin oikeaan paikkaan. Siten C-osoittimia ei tulisi välittää tai käyttää tilanteissa, joissa validointia tarvitaan. Tällaisiin tilanteisiin sopivat esimerkiksi olio-ohjelmointi ja menetelmät, jotka perustuvat tiedon kätkentään ja tyyppitarkistuksiin.

**Implementointisääntö 16 (Validointisääntö).** Suunnittele tietorakenteet ja funktiot siten, että ne voidaan validoida.

# 10. Virheellinen atomisuus

Virheellinen atomisuus tai jakamattomuus syntyy, kun operaatio on periaatteessa jakamaton, mutta se toteutetaan huolimattomasti kahdella operaatiolla.

**Implementointisääntö 17 (Atomisuussääntö).** Jos kaksi operaatiota täytyy suorittaa peräkkäin ilman, että mitään suoritetaan välissä, käytä mekanismeja, jotka takaavat, että näin todella tapahtuu.

# 11. Virheellinen järjestys

Tarkastellaan tilannetta, jossa prosessi luo lock-tiedoston ja sen jälkeen kirjoittaa log-tiedostoon. Toinen prosessi puolestaan kirjoittaa ensin log-tiedostoon ja tarkistaa sen jälkeen, onko lock-tiedosto olemassa. Ensimmäinen käyttää oikeaa järjestystä, toinen väärää (koska tällöin monet voivat kirjoittaa log-tiedostoon samanaikaisesti).

**Implementointisääntö 18 (Järjestyssääntö).** Määrittele resurssiin tai objektiin kohdistuvat lailliset operaatiot ja niiden järjestykset. Tarkista, että kaikissa ohjelman suorituksissa operaatioiden järjestys on sallittu.

# 12. Operaation tai operandien virheellinen valinta

Parametrien tulee olla sopivaa tyyppiä ja sopivan arvoisia ja operaatioiden tulee olla sallittujen funktioiden joukosta.

**Esimerkki.** UNIX-ohjelma su mahdollistaa, että käyttäjä saa toisen käyttäjän identiteetin. Tarinan mukaan eräs versio tästä ohjelmasta antoi käyttäjälle juurioikeudet, jos käyttäjätietokantaa ei ollut olemassa. Jos ohjelma ei voinut avata tietokantaa, se oletti, ettei sitä olekaan. Tämä oli virheellinen operaatio, sillä hyökkääjä voi estää pääsyn tietokantaan, vaikka se olisi olemassa. □

**Hallintosääntö 8 (Ohjelmstotuotannon menetelmien hallintosääntö).** Käytä ohjelmistotuotannon (software engineering) ja varmistuksen (assurance) tekniikoita (kuten kommentointia, suunitelmien ja koodin läpikäyntiä) varmistaaksesi, että operaatiot ja operandit ovat sopivia.

# Implementointisääntöjen ryhmittelyä I

**Helposti muistettavat ja toteutettavat säännöt**

- **Implementointisääntö 1 (Moduulien eristämissääntö)**. Strukturoi prosessi (ohjelma) siten, että ylimääräisiä oikeuksia tarvitsevat prosessin osat ovat omana moduulinaan. Moduulien tulisi olla niin pieniä kuin mahdollista ja niiden tulisi tehdä vain niitä tehtäviä, joissa noita ylimääräisiä oikeuksia tarvitaan.
- **Implementointisääntö 11 (Taulukkoviittausten sääntö).** Varmista, että kaikki taulukkoviittaukset kohdistuvat todella taulukkoon. Jos jonkun funktion kohdalla ei voi olla varma viittausrajoista, älä käytä kyseistä funktiota. Etsi toinen funktio, kirjoita uusi versio tai luo kääre (wrapper).

# Implementointisääntöjen ryhmittelyä II

- **Implementointisääntö 15 (Syötteiden tarkistussääntö).** Tarkista kaikki syöte sekä muodon että sisällön osalta. Erityisesti tarkista kokonaisluvuista, etteivät ne ole liian isoja tai pieniä, ja tarkista merkkitieto, että se on sopivan pituinen eikä sisällä muuta kuin sallittuja merkkejä.
- **Implementointisääntö 13 (Proseduurien tulosten seurantasääntö).** Tarkista kaikkien funktio- ja proseduurikutsujen yhteydessä, ettei kutsujen suorituksessa ole tapahtunut virheitä.
- **Implementointisääntö 14 (Sallittujen arvojen sääntö).** Tarkista, että muuttujan arvot ovat sallittuja.

**Hieman vaikeammin toteutettavat säännöt**

- **Implementointisääntö 3 (Muistinvarmistussääntö).** Varmista, ettei ohjelma jaa objekteja muistissa muiden ohjelmien kanssa ja että muut ohjelmat eivät pääse etuoikeutetun prosessin muistialueelle.

# Implementointisääntöjen ryhmittelyä III

- **Implementointisääntö 4 (Virhetilanteiden sääntö).** Jokaisen funktion käyttäytyminen virhetilanteissa täytyy selvittää. Älä yritä toipua, jos ei ole varmaa, etteivät seuraukset aiheuta turvallisuusongelmia. Ohjelman tulisi palauttaa systeemin tila samaksi kuin ennen alkua ja sitten lopettaa.
- **Implementointisääntö 6 (Poikkeuskäsittelijöiden sääntö).** Asynkronisten poikkeuskäsittelijöiden ei tulisi muuttaa mitään muita muuttujia kuin poikkeuskäsittelijän sen hetkisen moduulin lokaaleja muuttujia. Poikkeuskäsittelijän tulisi estää muut poikkeukset sen jälkeen kun poikkeus on tapahtunut, eikä estoa tulisi purkaa ennenkuin ko. poikkeuksen käsittely on päättynyt. Poikkeuksia tästä säännöstä voidaan tehdä vain, jos suunnittelussa otetaan huomioon poikkeukset poikkeusten sisällä.

# Implementointisääntöjen ryhmittelyä IV

- **Implementointisääntö 7 (Luotettavan datan sijoitussääntö).** Aina kun mahdollista, pidä prosessin luotettava data eri paikassa kuin data, joka tulee epäluotettavalta taholta. Huolehdi myös, että muistivirhe syntyy, jos epäluotettava data korvaa luotettavan.
- **Implementointisääntö 10 (Luottamuksellisten objektien tuhoamissääntö)**. Kun prosessi ei enää käytä luottamuksellista objektia, objekti tulisi puhdistaa ja sen jälkeen tuhota tai vapauttaa.
- **Implementointisääntö 12 (Tyyppien tarkistussääntö).** Tarkista funktioiden ja parametrien tyypit
- **Implementointisääntö 17 (Atomisuussääntö).** Jos kaksi operaatiota täytyy suorittaa peräkkäin ilman, että mitään suoritetaan välissä, käytä mekanismeja, jotka takaavat, että näin todella tapahtuu.

**Säännöt, joiden toteuttaminen vaikuttaa hankalalta**

# Implementointisääntöjen ryhmittelyä V

- **Implementointisääntö 2 (Oletuksien tarkistussääntö).** Varmista, että ohjelma tarkistaa aina käyttäessään muita tiedostoja tai ulkoisia resursseja, että niihin liittyvät oletukset ovat voimassa. Jos tämä ei ole mahdollista, dokumentoi tällaiset tilanteet ylläpitäjiä varten, jotta he tietävät mahdolliset hyökkäyskohteet.
- **Implementointisääntö 5 (Vuorovaikutuksien syknronointisääntö).** Jos prosessilla on vuorovaikutusta muiden prosessien kanssa, vuorovaikutukset tulee synkronoida. Erityisesti tulee tuntea kaikki erilaiset suorituspolut, joita vuorovaikutuksissa voi syntyä. Kaikkien suorituspolkujen tulee noudattaa valittua turvapolitiikkaa.
- **Implementointisääntö 8 (Epätoivottavien komponenttien sääntö) .** Älä käytä komponentteja, jotka voivat muuttua ohjelman ajojen välillä.

# Implementointisääntöjen ryhmittelyä VI

- **Implementointisääntö 9 (Kontekstisääntö).** Prosessin täytyy varmistaa, että konteksti, jossa objekti nimetään, määrittelee oikean objektin.
- **Implementointisääntö 16 (Validointisääntö).** Suunnittele tietorakenteet ja funktiot siten, että ne voidaan validoida.
- **Implementointisääntö 18 (Järjestyssääntö).** Määrittele resurssiin tai objektiin kohdistuvat lailliset operaatiot ja niiden järjestykset. Tarkista, että kaikissa ohjelman suorituksissa operaatioiden järjestys on sallittu.

# Ohjelmointikielten turvallisuuspiirteet I

- Ohjelmointikielten turvallisuuteen kuuluu kaksi eri asiaa. Ensinnäkin turvallinen kieli vähentää ohjelmointivirheiden, erityisesti ajoaikana vaikuttavien, mahdollisuutta.
- Virheitä on kahden tyyppisiä. Tietyt virheet aiheuttavat virheellistä toimintaa, joka voi johtaa systeemin romahtamiseen. Tämä on kiusallista tai suorastaan katastrofaalista, jos systeemi on esimerkiksi reaaliaikainen kriittinen systeemi kuten lentokoneen ohjelmisto tai ydinvoimalan valvontasysteemi.
- Tietyt virheet taas aiheuttavat sen, että hyökkääjä pääsee käsiksi systeemiin, mutta varsinaista virheellistä toimintaa ei välttämättä tapahdu.
- Virheiden välttämisen lisäksi turvallisen ohjelmointikielen tulisi tarjota erilaisia kryptografisia algoritmeja, joilla voidaan toteuttaa esimerkiksi salaus ja todennus.

# Ohjelmointikielten turvallisuuspiirteet II

- Monet ohjelmointikielten edistysaskeleista vaikuttavat virheiden vähenemiseen. Tällaisia edistysaskeleita ovat olleet modulaarisuus, olio-ohjelmointi ja poikkeukset. Toisaalta käytäntö ja sovellukset ovat pakottaneet ottamaan mukaan myös piirteitä, jotka edesauttavat virheiden syntymistä. Esimerkkinä tällaisista piirteistä on kommunikointi muiden koneiden kanssa.
- Eräs tärkeimmistä piirteistä turvallisuuden kannalta on tyypitys. Yleensä katsotaan, että *vahva tyypitys* karsii ajoaikaisia virhemahdollisuuksia. Vahva tyypitys tarkoittaa, että ohjelmointikielessä on tiukkoja rajoituksia, miten muuttujien arvoja voidaan käyttää. Esimerkiksi kokonaislukujen jakolaskua ei voi käyttää merkkijonojen yhteydessä. Toisin sanoen automaattisia tyypinmuunnoksia tapahtuu vain hyvin rajoitetusti.
- Vahva tyypitys voi sisältää seuraavia asioita (wikipedia: Strongly typed programming language):

# Ohjelmointikielten turvallisuuspiirteet III

- Vahvat takeet ohjelman käyttäytymisestä suorituksen aikana. Takeet saadaan staattisella analyysillä tms.
- Tyyppiturvallisuus (type safety). Tämä tarkoittaa, että käännös- tai ajoaikana hylätään operaatiot tai funktiokutsut, joissa ei ole välitetty tietotyypeistä.
- Takuu, että hyvin määritellyt virhe- tai poikkeusrutiinit laukeavat heti, kun tyyppien yhteensovittamisessa tulee ongelmia.
- Käännösaikainen tarkistus. Kääntäjä varmistaa, että operaatiot suoritetaan vain parametreilla, jotka noudattavat operaation tyypitystä.
- Dataobjektin tyyppi ei voi muuttua objektin elinaikana.
- Tyyppisysteemin koskemattomuus. Tyyppisysteemi ei ole koskematon kielissä, joissa on mahdollista muuttaa suoraan arvojen bittiesitystä.
- Implisiittisten tyypinmuunnosten puuttuminen. Ei ole mahdollista antaa kääntäjälle ohjeita tyypinmuunnoksista. Tyypinmuunnos on ainoastaan mahdollinen eksplisiittistä merkintää (cast) käyttämällä.

# Ohjelmointikielten turvallisuuspiirteet IV

- Toinen turvallisuuteen selvästi liittyvä ohjelmointikielten piirre on muistiviittaukset. Joissakin kielissä kuten C:ssä on osoittimet, jotka voivat viitata melko vapaasti muistiin. Tällainen mahdollisuus tekee kielestä joustavan ja moniin käyttöjärjestelmä- ja laiteläheisiin sovelluksiin sopivan kielen, mutta samalla se mahdollistaa vakavia virheitä.

# Java: hyvät puolet I

Java suunniteltiin aikana, jolloin web-sovellukset yleistyivät ja jolloin myös tietoturvaan alettiin kiinnittää entistä enemmän huomiota jo suunnitteluvaiheessa. Erityisesti vieraiden sovelmien lataamiseen ja suorittamiseen on kiinnitetty huomiota. Tähän ottaa osaa kolme moduulia: luokkalataaja, tavukoodin verifioija ja turvamanageri.

Luokkalataajan (class loader) tehtäviin kuuluu

- noutaa sovelman koodi kaukaiselta koneelta,
- luoda nimiavaruuden hierarkia,
- estää sovelmaa luomasta omaa luokkalataajansa ja
- estää sovelmaa kutsumasta metodeja, jotka kuuluvat systeemin luokkalataajalle.

# Java: hyvät puolet II

Nimiavaruuden hierarkian avulla varmistetaan, että suoritettavat sovelmat eivät syrjäytä systeemitason komponentteja ajoaikaisessa ympäristössä.

Ennenkuin sovelma voidaan suorittaa, tavukoodin verifioija verifioi sen. Verifioija varmistaa, että sovelman koodi noudattaa kaikkia kielen piirteitä. Itse asiassa verifioijan lähtöoletus on, että sovelman koodin tavoitteena on tuhota tai läpäistä systeemin turva-asetukset. Verifioija tekee seuraavat tarkistukset:

- Koodi on oikein muotoiltu.
- Sisäinen pino ei vuoda yli tai ali.
- Laittomia tyypinmuunnoksia ei tapahdu.
- Tavukoodin käskyillä on noikein tyypitetyt parametrit.
- Objektin yksityinen data jää yksityiseksi.

# Java: hyvät puolet III

Sovelman luokkalataaja luo uuden nimiavaruuden jokaista sovelmaa kohti. Sen tähden sovelmat voivat päästä käsiksi vain omiin luokkiinsa ja standardin Java-kirjaston API luokkiin. Ne eivät voi käyttää muiden sovelmien luokkia. Tällä on kaksi etua:

- Erillisten nimiavaruuksien johdosta sovelmien on vaikea tehdä yhteishyökkäystä systeemiä vastaan.
- Sovelmien kehittäjien ei tarvitse huolehtia nimien yhteentörmäyksistä. Nimen tulee olla yksikäsitteinen vain omassa nimiavaruudessaan.

On otettava huomioon kaksi heikkoutta. Ensinnäkin selainympäristöissä luokkalataaja on yleensä selaimen valmistajan tekemä. Tavallisesti se perustuu Sunin tekemille määrityksille, mutta käyttäjän täytyy joka tapauksessa luottaa valmistajaan. Toiseksi Java-sovellukset voivat luoda vapaasti omat luokkalataajansa. Tällöin on tietysti vaarana, ettei kaikkia turvallisuusnäkökohtia oteta huomioon.

# Java: hyvät puolet IV

Turvamanageri on kolmas ja tärkein osa Javan turvamallia. Sen tehtäviä ovat mm.

- hallita kaikkia pistokeoperaatiota (sockets),
- suojata pääsyä rajoitettuihin resursseihin,
- kontrolloida käyttöjärjestelmän ohjelmien lataamista ja yleensä niihin pääsyä,
- estää uusien luokkalataajien asennus,
- ylläpitää säikeiden eheyttä,
- kontrolloida pääsy Javan pakkauksiin.

Edellä mainitut asiat liittyvät pääsiassa muualta tuotujen sovelmien suorittamiseen. Sen sijaan Javan vahva tyypitys palvelee kaikkea ohjelmistokehitystä. Vahvan tyypityksen vuoksi kääntäjä estää , etteivät ohjelmat ja metodit käsittele muistia sopimattomalla tavalla. Javassa ei myöskään ole C:n kaltaisia osoittimia, joten muistiviittaukset ovat rajoitettuja.

# Java: huonot puolet

Javassa on kiinnitetty paljon huomiota turvallisuuteen. Turvallisuus (security) on kuitenkin eri asia kuin luotettavuus (reliability). Luotettava ohjelmointikieli on sellainen, jonka rakenne auttaa välttämään virheitä. Seuraavassa esitellään lähteeseen [**?**] perustuen muutamia Javan epäselvyyksiä, jotka eivät kuitenkaan yleensä johtane systeemiin tunkeutumiseen, vaan muuhun virheelliseen toimintaan.

# Esimerkki 1.

Tarkastellaan seuraavaa määrittelyä:

```
int var = 8262;
int far = 0250;
int bar = 9292;
```

Javassa on myös heksadesimaaliset ja oktaaliset kokonaisluvut. Luku 0250 onkin oktaaliluku (oktaaliluvut alkavat nollalla) ja sen arvo desimaalijärjestelmässä on 170. Tarkoittiko ohjelmoija todella näin? □

# Esimerkki 2.

Java on perinyt C:stä sijoitusmerkin $=$. Tarkastellaan koodia

```
boolean condition_1 = ...;
boolean condition_2 = ...;
...
if (condition_1=condition_2){
...
}
```

Onko ohjelmoija sekoittanut merkin $=$ merkin $==$ kanssa, vai onko hän todella tarkoittanut, että `condition_2` sijoitetaan `condition_1`:een ja lopputulosta käytetään if-lauseen ehtona. Tätä ei voi tietää tutkimatta koodia tarkemmin. Pascalissa ja Adassa sijoitus on $:=$, eikä sijoituksen tulosta voi "samaan aikaan" käyttää vertailussa. □

# Esimerkki 3. I

Javassa kuten C:ssä ohjelmoija voi jättää funktioiden paluuarvot huomiotta. Tarkastellaan seuraavaa koodia:

```
static int function_returning_an_int(int x) {...}

static void f() {
    int some_variable;
    int another_variable;
    ...
    another_variable = some_variable--;
    function_returning_an_int(1);
```

Selvästikin tässä on jotain virheellistä, mutta ohjelmoijan tarkoitusta ei ole helppo arvata eikä kääntäjä valita mistään. Oliko tarkoitus olla

# Esimerkki 3. II

```
some_variable--;
another_variable =
 function_returning_an_int(1);
```

vai

```
another_variable =
  some_variable-function_returning_an_int(1);
```

vai oliko todella tarkoitus jättää paluuarvo huomiotta? Joissakin kielissä, esimerkiksi Adassa, funktion paluuarvoa ei voi jättää huomiotta. Näissä kielissä esimerkin koodi tuottaisi kääntäjän virheilmoituksen ja virhe korjattaisiin heti. □

# Esimerkki 4. I

Erottimet {} ja ; voivat aiheuttaa sekaannusta. Onko seuraavassa ; `if`:in jälkeen tarkoitettu:

```
if (...); {
    ...
}
```

Onko seuraavassa ohjelmointivirhe:

```
if (...)
    x=1;
    y=2;
```

Adassa tällaista epämääräisyyttä ei ole, sillä if-lause täytyy päättää end if -komentoon. Toinen epämääräisyys liittyy else-rakenteeseen:

# Esimerkki 4. II

```
public static float sum_positive(float [] values,
                                 float default) {
   float sum =0.0;
   if (values.length > 0)
      for (int i=0; i<values.length; i++)
          if (values[i]>0.0)
             sum+=values[i];
          else
             sum=deafult;
          return sum;
}
```

Tässä else on sidottu toiseen if-lauseeseen, ei ensimmäiseen. Selvempää olisi, jos for-lauseella olisi loppusulku kuten Adassa end loop. □

# Esimerkki 5. I

Java käsittelee kokonaislukujen ylivuotoja käyttäen ns. wraparound-semantiikkaa. Siten

```
byte b = 127;
b++;
```

asettaa $b$:n arvoksi $-128$. Samanlaista tapahtuu, jos Javassa käytetään 16-, 32- tai 64-järjestelmän lukuja. Tällä on toisinaan yllättäviä sivuvaikutuksia kuten koodissa

```
public static void send_bytes_to_poprt (byte first,
                                        byte last) {
    for (byte b = first; b<= last; b++) {
      ...
    }
```

joka voi jäädä ikuiseen silmukkaan riippuen muuttujan `last` arvosta. □

# Esimerkki 6. I

Luokkien alustukset Javassa saattavat aiheuttaa ongelmia. Tarkastellaan koodia

```
public class A {
    static { System.out.println ("A begin"); }
    public static int x = B.g();
    public static int f () {
        System.out.println ("A.f()");
        return A.x + 1;
    }
    static { System.out.println ("A end"); }
}

public class B {
    static { System.out.println ("B begin"); }
```

# Esimerkki 6. II

```
    public static int z = 99;
    public static int y = A.f();
    public static int g () {
       System.out.println ("B.g()");
       return B.y + 1;
    }
    static { System.out.println ("B end"); }
}

public class C {
   public static void main (String [] args) {
      int i;
      if (args.length > 0)
          i = B.z;
      System.out.println ("A.x = " + A.x);
      System.out.println ("B.y = " + B.y);
```

# Esimerkki 6. III

```
        }
    }
```

Jos ohjelmaa kutsutaan ilman parametreja, tulostuu

```
A begin
B begin
A.f()
B end
B.g()
A end
A.x = 2
B.y = 1
```

Parametrien kanssa kutsuttaessa tulostuu puolestaan

# Esimerkki 6. IV

```
B begin
A begin
B.g()
A end
A.f()
B end
A.x = 1
B.y = 2
```

Javan luotettavuutta tällainen ei paranna. □

# Esimerkki 7. I

Vaikka Java on vahvasti tyypitetty kieli, se sallii kuitenkin joissakin tapauksissa implisiittiset tyypin muunnokset. Yhdessä kuormittamisen kanssa tämä voi aiheuttaa yllättäviä virheitä kuten seuraava koodi näyttää:

```
public class Try {
    static void p (float f)
      { System.out.println ("p (float)"); }
}

public class Client {
    public static void main (String args[])
       { Try.p (1); }
}
```

Ohjelmaa tulostaa `p (float)`. Oletetaan, että jonkin aikaa luokan Try kirjoittamisen jälkeen on tarvetta lisätä luokkaan toinen metodi p:

# Esimerkki 7. II

```
public class Try {
    static void p (float f)
      { System.out.println ("p (float)"); }
    static void p (int i)
      { System.out.println ("p (int)"); }
}
```

Jos nyt Client käännetään uudestaan ja ajetaan, se tulostaa `p (int)`. Tämä saattaa olla vakava tilanne, sillä luokan Try modifioija ei kenties tiedä edes luokan Client olemassaolosta. □

# Esimerkki 8. I

Javassa linkitys taphatuu dynaamisesti ohjelman ajoaikana ja on olemassa joukko sääntöjä, jotka pyrkivät varmistamaan, että muutokset ovat yhteensopivia kokonaisuuden kanssa. Lopputulos ei kuitenkaan aina ole luotettava.

```
public class Base { }
public class Deriv extends Base { }
public class Try {
     static void p (Base obj)
       { System.out.println ("p (Base)"); }
public class Client {
     public static void main (String args[])
       { Try.p (new Deriv ()); }
}
```

# Esimerkki 8. II

Jos ohjelma käännetään ja ajetaan, se tulostaa `p (Base)`. Jos luokkaa Try modifioidaan seuraavasti

```
public class Try {
    public static void p (Base obj)
        { System.out.println ("p (Base)"); }
    public static void p (Deriv obj)
        { System.out.println ("p (Deriv)");}
}
```

ja käännetään tiedosto Try.java, ohjelma tulostaa saman kuin äsken. Jos taas luokka käännetään uudelleen luokan Try muuttamisen jälkeen, tulostuu `p (Deriv)`. Hieman sekavaa! □

C ei ole vahvasti tyypitetty, vaan se sallii runsaasti automaattisia tyypinmuunnoksia. C:ssä on joustavat osoittimet, joiden huolimaton käyttö saattaa johtaa siihen, että hyökkääjä pääsee käsiksi asiaankuulumattomiin muistipaikkoihin. Seuraavassa on luettelo tärkeimmistä periaatteista, jotka täytyy ottaa huomioon C:llä ohjelmoitaessa.

- Tarkista kaikki tieto, joka luetaan ulkoisista lähteistä. Samoin kaikkien funktioiden tulisi tarkistaa parametreistansa, että ne ovat oletusten mukaisia. Ole erityisen huolellinen rajojen tarkistuksissa.
- Älä koskaan käytä funktiota `gets()` standardista syöttöjonosta lukemiseen. Käytä sen sijaan funktiota `fgets()`. Edellinen ei tarkista syötteen pituutta, mutta funktiossa `fgets()` voidaan antaa syötteen pituutta rajoittava parametri.
- Tarkista funktioiden paluuarvot. Monet systeemi- ja kirjastofunktiot palauttavat tiedon onnistumisestaan. Lopputulos tulisi aina tarkistaa.

# C II

- Älä koskaan käytä syötettä sellaisenaan formatoidussa merkkijonossa. Esimerkiksi `fprintf()` tulostaa annetun merkkijonon muodossa, joka vastaa toisena parametrina annettua formaattia. Jos esimerkiksi `stringpt` on osoitin syötteenä annettuun merkkijonoon, niin merkkijonoa ei pidä tulostaa komennolla `printf(ftp, stringpt)`, vaan turvallisemmalla tavalla `printf(ftp,"%s",stringpt)`.
- Vältä ympäristömuuttujien käyttöä, sillä niitä on helppo manipuloida. Jos niitä on pakko käyttää, niiden pituus tulee aina tarkistaa ja tulee myös tutkia, että muuttujien merkit kuuluvat sallittujen merkkien listaan.
- C:ssä voidaan ohjelmasta käynnistää muita ohjelmia ja prosesseja usealla eri tavalla. Vältä komentoja `system()` ja `open()` ja käytä sen sijaan komentoja `fork()` ja `exec()`.

# C III

- Jos ohjelma käsittelee luottamuksellista dataa, huolehdi, ettei se voi tuottaa muistivedosta (core dump). Tämä voidaan estää rajoittamalla muistovedoksen koko nollaan tavuun joko käyttämällä komentoa `ulimit(1)` ennen ohjelman ajoa tai kutsumalla funktiota `setrlimit()` ohjelman alussa.
- Käytä aina täydellisiä polkunimiä, kun avaat tiedostoja.
- Noudata pienimmän oikeuden periaatetta.
- Älä optimoi suorituskykyä kuin vasta koodin tarkistuksen jälkeen. Jos koodia optimoidaan alusta alkaen, tulos on vaikeasti ymmärrettävää.