OULUN AMMATTIKORKEAKOULU

Teea Põldsam

MUISTITURVALLISUUS OHJELMISTOKEHITYKSESSÄ TIETOTURVAN NÄKÖKULMASTA

MUISTITURVALLISUUS OHJELMISTOKEHITYKSESSÄ TIETOTURVAN NÄKÖ-KULMASTA

Teea Põldsam
Opinnäytetyö
Syksy 2024
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, Ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä: Teea Põldsam
Opinnäytetyön nimi: Muistiturvallisuus ohjelmistokehityksessä tietoturvan näkökulmasta
Työn ohjaaja: Pekka Alaluukas
Työn valmistumislukukausi ja -vuosi: Syksy 2024
Sivumäärä: 32

Muistiturvallisuus on ollut merkittävä, ajankohtainen ja olennainen aihe jo vuosikymmenien ajan. Tilastot osoittavat, että muistiturvallisuuteen liittyvistä ohjelmavirheistä johtuvat tietoturvahaavoittuvaisuudet ovat yksi suurin yksittäinen tietoturvariskiryhmä. Alkuvuodesta 2024 Yhdysvaltain Valkoinen talo julkaisi raportin, jonka aiheena on ohjelmistojen turvallisuuden parantaminen.

Tämän tutkivan opinnäytetyön aiheena oli muistiturvallisuus ja muistiturvalliset ohjelmointikielet tietoturvan näkökulmasta, muistiturvallisten ohjelmointikielten hyödyt ja haasteet sekä muistiturvallisempaan ohjelmistokehitykseen siirtyminen. Opinnäytetyön tavoitteena oli selkeyttää muistiturvallisuuden merkitystä ohjelmointikielen valinnassa sekä selvittää muistiturvallisempia vaihtoehtoja.

Opinnäytetyö toteutettiin analysoimalla tutkimuksia, raportteja ja tilastoja sekä perehtymällä muistiturvallisiin ja muistiturvattomiin ohjelmointikieliin käytännössä. Opinnäytetyö koostuu teoriasta, Valkoisen talon raportin ja eri yrityksien keräämien tilastojen analyysistä sekä käytännön esimerkeistä.

Opinnäytetyöstä käy ilmi, että muistiin liittyvät ohjelmavirheet ovat yleisiä, ja niistä voi seurata myös vakavia tietoturvariskejä. Muistiturvallisten ohjelmointikielten käyttöä kannattaisi harkita tilanteissa, joissa ei ole tarvetta suoralle yhteydelle muistinkäsittelyyn, ollaan aloittamassa uutta projektia tai joissa muistivirheet voivat aiheuttaa suuria ongelmia.

Suurien olemassa olevien ohjelmistojen uudelleenkirjoittaminen ei kuitenkaan ole helppoa eikä usein taloudellisesti kannattavaa, jolloin voidaan panostaa muihin keinoihin saavuttaa muistiturvallisempi koodi. Yksi kattava tapa parantaa muistiturvallisuutta ilman koko ohjelmiston uudelleenkirjoittamista on ohjelmistokehittäjien kouluttaminen tunnistamaan, korjaamaan ja ehkäisemään muistivirheitä.

Asiasanat: muistiturvallisuus, muistivirhe, Rust, Ada, C++, C

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author: Teea Põldsam
Title of thesis: Memory safety in software development from an information security perspective
Supervisor: Pekka Alaluukas
Term and year when the thesis was submitted: Autumn 2024
Number of pages: 32

Memory safety has been a relevant topic for decades. Statistics show that information security vulnerabilities resulting from memory safety bugs are one of the largest information security risk groups. In early 2024, The White House published a report about improving software security by developing memory safe code.

The topic of this thesis was memory safety and memory safe programming languages from a security perspective, the benefits and challenges of memory safe programming languages, and the transition to memory safe software development. The aim of this thesis is to clarify the importance of memory safety in choosing a programming language and to explore memory safety.

Memory bugs are common and can lead to serious information security risks. The use of memory safe programming languages should be considered in situations where there is no need for direct access to memory processing, when a new project is being started, or when memory errors can cause major problems.

However, rewriting large existing software is not an easy task and often not economically viable, so other means can be used to achieve more memory safe code. One comprehensive way to improve memory safety without rewriting the entire software is to train developers to identify, fix, and prevent memory errors.

Keywords: memory safety, memory errors, Rust, Ada, C++, C

SISÄLLYS

1 Johdanto

Muistiturvallisuus on ollut merkittävä, ajankohtainen ja olennainen aihe jo vuosikymmenien ajan. Yksi ensimmäisistä internetin kautta levinneistä haittaohjelmista oli vuonna 1988 yli 6000 tietokoneeseen levinnyt Morris-mato, mikä tarkoitti, että noin 10 % kaikista siihen aikaan internettiin kytketyistä tietokoneista sai tartunnan (FBI 2018, Harán 2018). Morris-madon kehittäjä hyödynsi erilaisia haavoittuvaisuuksia, joista yksi oli puskurin ylivuoto (engl. buffer overflow), joka on eräänlainen muistiturvallisuuteen liittyvä ohjelmavirhe (Vu 2019).

Useat tilastot osoittavat, että muistiturvallisuuteen liittyvistä ohjelmavirheistä johtuvat tietoturvahaavoittuvaisuudet ovat yksi suurin yksittäinen tietoturvariskiryhmä. Esimerkiksi Googlen ohjaaman Chromium projektin tilastojen mukaan noin 70 % heidän projektinsa vakavista tietoturvaohjelmavirheistä ovat muistiturvallisuuteen liittyviä ongelmia (The Chromium Projects 2024).

Muistiturvallisuuteen liittyvät ohjelmavirheet ja niistä johtuvat tietoturvariskit ovat herättäneet huomiota jopa valtion tasolla asti. Vuonna 2024 Yhdysvaltain Valkoinen talo julkaisi raportin, jonka aiheena on ohjelmistojen turvallisuuden parantaminen. Raportissa korostetaan erityisesti ohjelmistojen ja ohjelmointikielien muistiturvallisuutta. Raportin mukaan siirtymällä muistiturvallisiin ohjelmointikieliin, voitaisiin estää kokonaisia tietoturvahaavoittuvuuksien ryhmiä. (The White House 2024a.)

Tämän tutkivan opinnäytetyön aiheena on muistiturvallisuus ja muistiturvalliset ohjelmointikielet tietoturvan näkökulmasta, muistiturvallisten ohjelmointikielten hyödyt ja haasteet sekä muistiturvallisempaan ohjelmistokehitykseen siirtyminen. Opinnäytetyön tavoitteena on selkeyttää muistiturvallisuuden merkitystä ohjelmointikielen valinnassa sekä selvittää muistiturvallisempia vaihtoehtoja ohjelmoijille, yrityksille ja organisaatioille.

Koska muistiturvallisuus ja muistiturvalliset ohjelmointikielet ovat olleet merkittäviä keskustelun aiheita hyvin pitkään, aiempia tutkimuksia ja tilastoja, joita voidaan hyödyntää tässä opinnäytetyössä, löytyy kohtalaisesti. Opinnäytetyö toteutetaan analysoimalla tutkimuksia, raportteja ja tilastoja sekä perehtymällä muistiturvallisiin ja muistiturvattomiin ohjelmointikieliin käytännössä. Opinnäytetyö koostuu teoriasta, Valkoisen talon raportin ja eri yrityksien keräämien tilastojen analyysistä sekä

käytännön esimerkeistä. Näiden monipuolisten lähestymistapojen ansiosta opinnäytetyö tarjoaa kattavan näkökulman aiheeseen.

2 MUISTITURVALLISUUS

Muistiturvallisuus (engl. memory safety) tarkoittaa muistin käyttöön ja hallintaan liittyvien ohjelmavirheiden puuttumista (Prossimo 2024). Tarkemmin määriteltynä ongelma on muistiin liittyvistä ohjelmavirheistä johtuvat haavoittuvaisuudet, joita hyökkääjät pystyvät hyödyntämään tietoturvahyökkäyksissä (Azevedo de Amorim, Hriţcu & Pierce 2018). Puutteellisesta muistiturvallisuudesta seuraa usein erilaisia tietoturvariskejä. Toisin sanoen kaikki muistiin liittyvät ohjelmavirheet estämällä voitaisiin poistaa kokonainen ryhmä tietoturvariskejä.

Tässä opinnäytetyössä keskitytään muistiin liittyvien ohjelmointivirheiden aiheuttamiin tietoturvariskeihin, mutta muistivirheet vaikuttavat myös ohjelmien vakauteen. Muistivirheistä voi seurata ohjelmistojen epävakautta, kuten kaatumisia, mikä johtaa tietoturvariskien lisäksi myös heikentyneeseen käyttökokemukseen (Prossimo 2024). Toisin sanoen muistiin liittyvät ohjelmavirheet heikentävät siis sekä tietoturvaa että ohjelman toimivuutta.

2.1 Muistiturvalliset ohjelmointikielet

Ohjelmointikielissä muistiturvallisuudella tarkoitetaan eri ominaisuuksia ja käytänteitä, jotka varmistavat, että muistia käsitellään ja käytetään oikein. Muistiturvalliset ohjelmointikielet estävät ohjelmoijaa tekemästä muistin käyttöön liittyviä ohjelmavirheitä niiden erilaisilla muistiturvallisilla ominaisuuksillaan. Jokainen muistiturvallinen ohjelmointikieli, kuten Java, C#, Rust ja Ada, toteuttaa muistiturvallisuuden hieman eri tavoin. Muistiturvallisuuden perustana on kuitenkin muistinhallinnan vastuun siirtäminen ohjelmoijalta ohjelmointikielelle. (Prossimo 2024.)

Vastuun siirtämisellä tarkoitetaan, että ohjelmointikieli huolehtii muistinhallinnasta automaattisesti, jolloin ohjelmoijan ei tarvitse huolehtia muistin varaamisesta tai vapauttamisesta, eikä millään tavoin itse valvoa muistivirheitä manuaalisesti (Fernando 2023). Kun vastuu turvallisesta muistinkäytöstä ei jää ohjelmoijalle, koodista ja ohjelmistoista tulee turvallisempia, koska inhimilliset muistivirheet katoavat.

Muistiturvallisissa ohjelmointikielissä muistiin liittyvät ohjelmavirheet estetään yleensä ensisijaisesti kielen rakenteiden ja ajonaikaisen ympäristön avulla, joten virheitä ei yleensä edes pääse syntymään. Jos muistiongelmia kuitenkin esiintyisi, ne havaittaisiin tyypillisesti kääntäjän tuottamina virheilmoituksina tai ohjelman ajonaikaisten poikkeusten kautta. (Prossimo 2024.)

Esimerkiksi Java- ja C#-ohjelmointikielet saavuttavat muistiturvallisuuden ajonaikaisen ympäristön, virtuaalikoneen, avulla. Molemmat kielet, kuten useat muutkin muistiturvalliset ohjelmointikielet, hyödyntävät automaattista roskankeruuta, joka poistaa tarpeettomat tiedot muistista. Tämä vähentää muistinhallinnan tarvetta ja ohjelmointivirheitä, mutta toisaalta roskankeruun ajonaikaiset viiveet voivat aiheuttaa ohjelmistojen epävakautta. (Blue Coding 2023.)

On myös muistiturvallisia ohjelmointikieliä, jotka eivät käytä roskankeruuta eikä virtuaalikonetta, kuten Rust, jossa muistiturvallisuus saavutetaan käännösaikaisilla tarkistuksilla. Tämän ansiosta Rust-ohjelmistot ovat usein kevyempiä ja toiminnaltaan ennakoitavampia kuin esimerkiksi Java- ja C#-ohjelmistot. (Ochoa 2024.)

Toisaalta tämä muistinhallinnan vastuun siirtäminen ohjelmoijalta ohjelmointikielelle tarkoittaa myös vapauden rajoittamista. Ohjelmoijan vapaus hallita muistia suoraan on rajoitettu, koska kieli automatisoi muistinhallinnan ja karsii muistin hallintaan liittyviä mahdollisuuksia (Fernando 2023). Tämä tarkoittaa, että ohjelmoijalla on vähemmän joustavuutta esimerkiksi muistin hienosäätöön ja suorituskyvyn optimointiin.

2.1.1 Rust

Graydon Hoaren suunnittelema ja Mozillan sponsoroima Rust-ohjelmointikieli on yksi esimerkki muistiturvallisista ohjelmointikielistä. Rust-kielen viime vuosina saavuttama suosio perustuu moniin tekijöihin. Sitä voidaan käyttää monipuolisesti eri tarkoituksiin, niin web- ja mobiilisovelluksissa kuin sulautetuissakin järjestelmissä. Rustin jatkuvasti kasvava yhteisö sekä kattava dokumentaatio tekevät siitä houkuttelevan vaihtoehdon myös aloitteleville ohjelmoijille (Codex 2023). Lisäksi monet suuret ja tunnetut yritykset, kuten Microsoft, Amazon ja Dropbox, ovat jo ottaneet Rustin käyttöön, mikä tarkoittaa, että kielen osaajille on kysyntää. Kuitenkin Rust on erityisesti tunnettu muistiturvallisuudestaan, sillä sitä suositellaan turvallisuutensa vuoksi jopa valtion tasolla (The White House 2024b). (Thompson 2023.)

Rustin muistiturvallisuus perustuu sen erilaisiin rakenteellisiin ominaisuuksiin, joiden ansiosta yleiset muistivirheet estetään jo käännösaikana ilman automaattista roskankeruuta. Näitä ominaisuuksia ovat esimerkiksi Rustin omistajuusmalli (engl. ownership) ja lainaamissäännöt. (Rust 2024a.)

Omistajuusmalli tarkoittaa, että jokaisella tietorakenteella on vain yksi selkeä omistaja, joka hallinnoi sitä ja sen elinkaarta. Omistajuusmallissa on kolme sääntöä:

1. Jokaisella arvolla on yksi omistaja.
2. Omistajia voi olla vain yksi.
3. Kun omistaja siirtyy soveltamisalueen ulkopuolelle, siihen liittyvä arvo vapautetaan automaattisesti.

Tämä estää kaksinkertaisen muistinvapautuksen (engl. double free) ja riippuva osoitin –ongelmat (engl. dangling pointer), joissa viitataan vapautettuun muistiin. (Rust 2024b.)

Rustissa lainaaminen tarkoittaa, että muuttujaa voidaan lainata muuttumattomana (engl. immutable borrow) tai muokattavana (engl. mutable borrow). Muuttumattomana lainaaminen tarkoittaa, että muuttujaan voidaan viitata useasta eri paikasta, mutta muuttujan arvoa ei voida muuttaa. Muokattavana lainaaminen tarkoittaa, että tietoon voidaan viitata ja sitä voidaan muokata vain yhdestä paikasta kerrallaan. Tämä ominaisuus estää ristiriitaiset muutokset ja virheet, jotka liittyvät tietojen samanaikaiseen lukuun ja kirjoittamiseen. (Rust 2024c.)

Rustin kääntäjä tarkistaa koodin käännösaikana, mikä varmistaa ohjelman muistiturvallisuuden, koska muistivirheet havaitaan ennen kuin ohjelma edes ajetaan. Ohjelmoija ei voi kiertää näitä sääntöjä, ellei koodia merkitä epäluotettavaksi ”unsafe”-avainsanalla. (Rust 2024d.)

Unsafe Rust sallii ohjelmoijan ohittaa Rustin normaalit turvallisuustakuut. Tietyissä tapauksissa ohjelmoija voi haluta toteuttaa operaatioita, jotka eivät ole turvallisia, kuten viitata suoraan muistiosoitteisiin tai kutsua muistiturvatonta koodia. Unsafe-avainsana ei kuitenkaan poista käytöstä Rustin turvallisuustarkastuksia, se vain mahdollistaa muistiturvattomien toimintojen toteuttamisen. Näin myös muistiturvaton koodi on rajattu yhteen alueeseen, jolloin ongelmat on helpompi löytää. (Rust 2024d.)

2.1.2 Ada

Ada-ohjelmointikieli on ISO-standardoitu muistiturvallinen ohjelmointikieli, joka kehitettiin alkujaan 1980-luvulla Yhdysvaltain puolustusministeriön tilauksesta korvaamaan useita satoja ohjelmointikieliä. Ada suunniteltiin turvalliseksi, tehokkaaksi ja tarkaksi ohjelmointikieleksi esimerkiksi avaruus-, ilmailu- ja sotilasjärjestelmiin. Adan ISO- standardi, ISO/IEC 8652, määrittelee Adalle tarkat vaatimukset, jotka mahdollistavat sen käytön turvallisuuskriittisissä järjestelmissä. (University of Michigan 2024, ISO/IEC 8652:2023.)

Ada on muistiturvallinen ohjelmointikieli erilaisten sisäänrakennettujen ominaisuuksien ansiosta, joiden seurauksena ohjelmoija ei voi vahingossa käyttää muistia väärin. Näitä ominaisuuksia ovat esimerkiksi automaattinen muistinhallinta ja vahva tyyppijärjestelmä. (Ada 2024.)

Adan vahva tyyppijärjestelmä on yksi kielen tunnetuimmista ominaisuuksista. Kielen tarkka tyyppimäärittely pakottaa ohjelmoijan määrittelemään tarkasti, minkä tyyppistä tietoa muuttujat voivat sisältää ja miten niitä voidaan käsitellä. Mahdolliset tyyppivirheet havaitaan käännösaikana. Tämä ehkäisee yhteensopimattomien tietotyyppien käytön ja tekee ohjelmasta helpommin ylläpidettäviä ja luotettavampia. (Ada 2024.)

2.2 Muistiturvattomat ohjelmointikielet

Muistiturvattomat ohjelmointikielet, kuten C ja C++, eivät tarjoa sisäänrakennettuja mekanismeja, jotka estäisivät muistinhallintavirheitä. Tästä seuraa, että ohjelmoija voi vahingossa kirjoittaa erilaisia muistiturvallisuuteen liittyviä ohjelmavirheitä. Eikä virheitä välttämättä löydetä edes ohjelmien käännös- tai suoritusaikana, jolloin ohjelmavirheet päätyvät ohjelmien lopullisiin versioihin ja saattavat aiheuttaa suuria tietoturvariskejä. (Gregory 2024.)

Muistiturvattomissa ohjelmointikielissä muistinhallinnan vastuu on jätetty kokonaan ohjelmoijalle, mistä voi seurata erilaisia muistivirheitä. Yleisiä muistivirheitä ovat esimerkiksi muistivuodot (engl. memory leak), riippuvat osoittimet (engl. dangling pointer), vapautuksen jälkeinen käyttö (engl. use-after-free) ja kaksoisvapautukset (engl. double free) (Granot 2024). Muistivirheitä käsitellään tarkemmin kappaleessa 3.

2.2.1 C++

Yksi tunnetuimmista muistiturvattomista ohjelmointikielistä on C++ (Statista 2024). Kyseessä on Bjarne Stroustrupin kehittämä ohjelmointikieli, jonka ensimmäinen versio julkaistiin jo vuonna 1985. C++ kehitettiin korvaamaan vanhempia, epäkäytännöllisiä ohjelmointikieliä. C++-ohjelmointikielestä tuli nopeasti suosittu erityisesti järjestelmäohjelmoinnin, peliohjelmoinnin ja sulautettujen järjestelmien kehityksessä (Snyk 2024a). (GeeksforGeeks 2022.)

C++-ohjelmointikielen vahvuuksiin kuuluvat tehokkuus, joustavuus ja suorituskyky, jotka ovat pitkään houkutelleet ohjelmoijia, yrityksiä ja organisaatioita valitsemaan sen pohjaksi vaativille sovelluksille. Erityisesti viime vuosina kieli on kuitenkin saanut paljon kritiikkiä sen muistiturvallisuuden puutteista (The White House 2024b). C++ ei tarjoa sisäänrakennettuja muistiturvaominaisuuksia, vaan muistinhallinta on jätetty ohjelmoijan vastuulle, mikä tekee siitä muistiturvattoman kielen, mutta tarjoaa taas vapautta ohjelmoijalle. C++ sallii suoran pääsyn muistipaikkoihin osoittimien avulla, mikä lisää tehokkuutta, mutta myös merkittäviä riskejä. (Deronjic 2023.)

Muistiturvallisuusongelmista huolimatta C++ on yksi käytetyimmistä ohjelmointikielistä. Kieltä on myös käytetty jo useita kymmeniä vuosia, mikä tarkoittaa, että osa järjestelmistä voi olla hyvinkin vanhoja (Deronjic 2023).

Uudemmissa C++ standardeissa (esim. C++11 ja uudemmat) on otettu käyttöön työkaluja, kuten älykkäät osoittimet (engl. smart pointer), jotka helpottavat muistinhallintaa ja siten vähentävät muistivuotojen riskiä. Näiden työkalujen avulla muistinhallinta automatisoituu osittain, mutta asianmukaisen muistinhallinnan vastuu on silti yhä ohjelmoijalla. (Microsoft 2021.)

The C++ Alliance ja Sean Baxter työstävät Safe C++ laajennusta, joka lisäisi muistiturvallisuuteen liittyviä ominaisuuksia C++:aan (Falco 2024). Tässä vaiheessa jää vielä nähtäväksi, minkälainen laajennus siitä tulee, mutta onnistuessaan kyseinen laajennus voi avata uuden, muistiturvallisemman sivun C++-ohjelmointikielen historiassa.

2.2.2 C

C-ohjelmointikieli on Dennis Ritchien 1970-luvulla kehittämä ohjelmointikieli, jonka alkuperäinen tarkoitus oli ratkaista aikaisempien ohjelmointikielien ongelmia. C toi merkittäviä edistysaskelia modernia ohjelmistokehitystä kohti ja monet ohjelmointikielet, kuten C++ ja Java, ovatkin ottaneet vaikutteita siitä. Vaikka nykyään onkin paljon moderneja ohjelmointikieliä, joista valita, C on yhä suosituimpien ohjelmointikielien joukossa. (Jonna 2024.)

C-ohjelmointikielellä on yhä paljonkin etuja. Yksi C:n vahvuuksista on sen siirrettävyys, mikä tarkoittaa, että C:llä kirjoitetut ohjelmat toimivat eri laitteissa ilman mitään muutoksia tai hyvin vähäisillä muutoksilla. C-kielellä ohjelmointia helpottaa myös laajat kirjastot ja vanha koodikanta, joita ohjelmoijat pystyvät hyödyntämään kehitystyössä. (Jonna 2024.)

C-ohjelmointikieli tarjoaa täyden pääsyn muistinhallintaan, mikä toisaalta tarkoittaa, ettei se sisällä muistiturvallisuusominaisuuksia. Tämä tarkoittaa, että muistin vapauttaminen ja jakaminen jää ohjelmoijan vastuulle. C on merkittävä tietoturvariskien lähde, sillä erityisesti muistiturvallisuuteen liittyviä ohjelmavirheitä syntyy paljon. (Jonna 2024.)

3 YLEISIÄ MUISTIVIRHEITÄ

Muistin käyttöön ja hallintaan liittyviä ohjelmavirheitä on monenlaisia riippuen käytetystä muistiturvattomasta ohjelmointikielestä. Monet C- ja C++-ohjelmointikielellä kirjoitetuista muistivirheistä ovat niin yleisiä, että virheillä on omat nimet (Harvard 2021). Näitä yleisiä muistivirheitä ovat esimerkiksi muistivuodot, riippuvat osoittimet, vapautuksen jälkeinen käyttö ja kaksoisvapautukset.

3.1 Muistivuodot

Muistivuoto (engl. memory leak) tapahtuu, kun ohjelmoija varaa muistia ohjelman käyttöön, muttei vapauta sitä, kun muistia ei enää tarvita (kuva 1.). Tästä seuraa, ettei varattu muisti koskaan palaudu järjestelmän käyttöön muille prosesseille. Jos muistivuotoja kertyy paljon tai ohjelmaa ajetaan pitkään, muisti voi täyttyä ja lopulta loppua kokonaan. Tämä hidastaa ohjelmaa ja voi jopa aiheuttaa sen kaatumisen, kun muistia ei enää ole saatavilla. Muistiturvallisissa ohjelmointikielissä muistivuodoilta vältytään esimerkiksi automaattisen roskankeruun ansiosta, joka huolehtii muistin vapauttamisesta ohjelmoijan puolesta (Diaz 2023). (GeeksforGeeks 2024.)

Koodi alkaa.

```
void leak() {
    int* number = new int(42); // Varataan muistia
    // Ei vapauteta muistia
}

int main() {
    while (true) {
        leak(); // Jokaisella kutsulla lisää muistia "vuotaa"
    }
}
```

Koodi päättyy.

KUVA 1. Yksinkertaistettu muistivuoto C++-ohjelmointikielellä

Muistivuotoja hyödynnetään tietoturvahyökkäyksissä. Muistivuodot tekevät ohjelmista epävakaita, tietoja voi kadota ja mahdollinen hyökkääjä voi pystyä manipuloimaan muistivuotoja omiin tarkoituksiin. (ReasonLabs 2023.)

3.2 Riippuvat osoittimet

Riippuva osoitin (engl. dangling pointer) syntyy, kun osoitin viittaa muistialueeseen, joka on jo vapautettu tai poistunut käytöstä (kuva 2). Toisin sanoen osoitin ei viittaa enää mihinkään käyttökelpoiseen muistiin. Tällaisen osoittimen käyttö voi johtaa odottamattomiin virheisiin tai ohjelman kaatumiseen. (GeeksforGeeks 2024b.)

Koodi alkaa.

```
int* dangling_pointer() {
    int* number = new int(42); // Varataan muistia
    delete number; // Vapautetaan muisti
    return number; // Palautetaan osoitin vapautettuun muistiin
}

int main() {
    int* ptr = dangling_pointer();
    std::cout << *ptr << std::endl; // Käytetään jo vapautettua muistia
    return 0;
}
```

Koodi päättyy.

KUVA 2. Yksinkertaistettu riippuva osoitin C++-ohjelmointikielellä

Jos ohjelmoija kuvittelee kuvan 2 ohjelman tulostavan arvon 42, lopputulos onkin yllättävä. Koska muisti on jo vapautettu, ohjelma tulostaakin yllättäviä arvoja. Tästä voi seurata epävakautta, kaatumisia ja datan vioittumista. Riippuvan osoittimen takia tietoturvahyökkääjä voi pystyä päästä käsiksi arkoihin tietoihin. (GeeksforGeeks 2024b.)

3.3 Käyttö vapautuksen jälkeen

Käyttö vapautuksen jälkeen (engl. use-after-free) tarkoittaa, että muistialuetta yritetään käyttää sen jälkeen, kun se on jo vapautettu (kuva 3). Tästä seuraa odottamattomia tuloksia, koska vapautettu muisti voi olla jo annettu uudelleen jollekin toiselle osalle ohjelmaa. Käyttö vapautuksen jälkeen on yleinen tietoturvaongelma, koska hyökkääjät voivat joskus manipuloida vapautettua muistia ja käyttää sitä väärin. (Snyk 2024b)

Koodi alkaa.

```
int main() {
  int* number = new int(42); // Varataan muistia
  std::cout << "Number: " << *number << std::endl;

  delete number; // Vapautetaan muisti
  std::cout << "Number after delete: " << *number << std::endl;

  return 0;
}
```

Koodi päättyy.

KUVA 3. Yksinkertaistettu käyttö vapautuksen jälkeen C++-ohjelmointikielellä

Kuvan 3 ohjelma tulostaa ensin arvon 42, mutta muistin vapauttamisen jälkeen ohjelma tulostaa yllättäviä arvoja. Näiden muistivirheiden estämiseen on erilaisia työkaluja, joista yksi on modernin C++-ohjelmointikielen älykkäiden osoittimien käyttö (Snyk 2024b). Muistiturvallisten ohjelmointikielien erilaiset rakenteet ja tarkistukset varmistavat, että vapautettua muistia ei käytetä uudelleen, jolloin tällaisia virheitä ei tapahdu.

3.4 Kaksoisvapautus

Kaksoisvapautus (engl. double free) tarkoittaa, että sama muistialue yritetään vapauttaa kahdesti (kuva 4). Kaksoisvapautus voi olla tietoturvariski, koska hyökkääjät voivat yrittää manipuloida muistia ohjelman kaatamiseksi tai suorittamiseksi hallitsemattomassa tilassa. (Kapil 2020.)

Koodi alkaa.

```
int main() {
  int* number = new int(42); // Varataan muistia
  std::cout << "Number: " << *number << std::endl;

  delete number; // Vapautetaan muisti
  delete number; // Vapautetaan muisti

  return 0;
}
```

Koodi päättyy.

KUVA 4. Yksinkertaistettu kaksoisvapautus C++-ohjelmointikielellä.

Isommissa ohjelmissa kaksoisvapautusvirheistä voi seurata kaatumisia tai muunlaista odottamatonta käytöstä (Kapil 2020). Tällaisia virheitä ei tapahdu muistiturvallisilla ohjelmointikielillä ohjelmoidessa, sillä niiden erilaiset rakenteet ja tarkistukset varmistavat, ettei muistia vapauteta uudelleen, jolloin tällaisia virheitä ei tapahdu.

4 MUISTITURVALLISUUS TILASTOISSA

Monet yritykset ja organisaatiot, kuten Microsoft ja Google, keräävät kattavia tilastoja ohjelmavirheistä ja tietoturvariskeistä sekä niiden alkuperistä. Useat tilastot osoittavat, että muistiturvallisuuteen liittyvät ongelmat ovat erittäin yleisiä tietoturvariskejä. Esimerkiksi Microsoftin tutkimukset osoittavat, että ongelma on erityisesti muistiturvattomissa ohjelmointikielissä (kuten C ja C++), jotka eivät tarjoa rakenteellisia muistiturvamekanismeja (Microsoft 2019a).

Microsoftin tutkimusten mukaan noin 70 % heidän kaikista tietoturvahaavoittuvuuksista liittyy muistivirheisiin. Microsoft raportoi, että nimenomaan muistivirheet ovat yksi suurimmista tietoturvariskeistä heidän tuotteissansa. Microsoft on viime vuosina investoinut turvallisempiin ohjelmointikieliin, kuten Rustiin (Microsoft 2019b, Microsoft 2019a). Microsoft raportoi myös, että vaikka C++ ei ole muistiturvallinen, muistiturvallisuutta saa lisättyä käyttämällä modernia C++-versiota ja sen ominaisuuksia kuten älykkäitä osoittimia (engl. smart pointer) (Microsoft 2019c).

Googlen vastaavat tutkimukset osoittavat, että suurin osa heidän Android- ja Chrome-projektien tietoturvahaavoittuvuuksista johtuu muistivirheistä. Googlen tietojen mukaan jopa 90 % Chrome-selaimen kriittisistä haavoittuvuuksista johtuu muistinhallintavirheistä. Tästä seurauksena Google on ottanut käyttöön muistiturvallisia ohjelmointikieliä, kuten Rustin, erityisesti uusissa Chrome-projekteissa ja Android-kehityksessä. (Rebert & Kern 2024.)

Mozillalla on pitkää kokemusta muistiturvallisuusongelmien kanssa. Esimerkiksi Mozillan Firefox-selaimen kehityksessä käytettiin paljon C++ koodia. Mozillan raporttien mukaan muistivirheet ovat olleet suurin yksittäinen syy tietoturvahaavoittuvaisuuksiin ja jopa 70 % kriittisistä haavoittuvaisuuksista Firefox-selaimessa, mistä seurasi Rustin laajempi käyttöönotto (esimerkiksi Servo-renderöintimoottorissa). Rustin käyttö vähensi merkittävästi uusien muistiturvaan liittyvien virheiden riskiä ja Firefoxin tietoturvakorjausten määrää. (Anderson ym 2015.)

Myös Amazon on julkaissut näkemyksiä muistiturvallisuudesta, erityisesti AWS-palveluiden kehityksessä. He ovat todenneet, että muistivirheet muodostavat riskin suurten palvelinjärjestelmien vakaudelle ja tietoturvalle. Amazon on tämän takia ottanut Rustin käyttöön AWS ydinkomponenteissa (esim. Firecracker-virtuaalikoneessa). Rustin käyttäminen lisäsi muistiturvallisuutta ja vähensi tietoturvariskejä ilman suorituskyvyn heikkenemistä. (Asay 2020.)

Monet suuret teknologiayritykset ovat alkaneet tutkia muistiturvallisuutta ja ottaa käyttöön muistiturvallisia ohjelmointikieliä. Monet yritykset, kuten Mozilla ja Amazon, ovat onnistuneet vähentämään tietoturvariskejä vaihtamalla edes osan käytetyistä ohjelmointikielistä muistiturvalliseen vaihtoehtoon, kuten Rustiin. Yhä useammat yritykset ja organisaatiot ovat kiinnostuneita muistiturvallisuudesta ja muistiturvallisista ohjelmointikielistä, sillä ne vähentävät tarvetta jatkuville muistivirheiden tarkastuksille ja ohjelmakorjauksille. Pitkällä aikavälillä muistiturvallisten ohjelmointikielten käyttäminen on siis myös taloudellisesti kannattavaa.

5 YHDYSVALTAIN VALKOISEN TALON NÄKEMYS MUISTITURVALLISUUDESTA

Helmikuussa vuonna 2024 Yhdysvaltain Valkoinen talo julkaisi raportin, "**Back To The Building Blocks: A Path Toward Secure And Measurable Software**" (suom. Takaisin rakennuspalikoihin: polku kohti turvallista ja mitattavaa ohjelmistoa), joka käsittelee ohjelmistojen turvallisuuden parantamista. Raportin keskeisenä aiheena on muistiturvallisuus osana turvallista ohjelmistokehitystä. (The White House 2024a.)

Erityisesti raportissa painotetaan muistiturvattomista ohjelmointikielistä, kuten C ja C++, siirtymistä muistiturvallisiin ohjelmointikieliin, kuten Rustiin. Raportin mukaan muistiturvallisten ohjelmointikielten käyttäminen toisi mukanaan useita hyötyjä paremman tietoturvan lisäksi. Esimerkiksi muistiturvallisuuden saavuttaminen tapahtuu tehokkaammin, kun se peritään ohjelmointikielten rakenteista eikä muistivirheitä tarvitse korjata jälkikäteen. Tästä seuraa myös sekä ohjelmistokehityksen tehokkuuden paraneminen, koska kehittäjien ei tarvitse käyttää niin paljon aikaa muistivirheiden etsimiseen ja korjaamiseen, että ylläpidon helpottuminen, koska turvallisuuspäivitysten ja korjausten tarve vähenee. Raportti tukee muistiturvallisten ohjelmointikielten laajempaa käyttöönottoa ja suosittelee kielten opettamista, käyttöä ja integrointia erityisesti tietoturvakriittisiin projekteihin ja järjestelmiin. (The White House 2024b.)

Muistiturvallisten ohjelmointikielten käyttämisen lisäksi raportissa mainitaan muita keinoja lisätä muistiturvallisuutta. Raportissa korostetaan koodin tarkastamisen ja testaamisen tärkeyttä, jotta mahdolliset muistivirheet voidaan tunnistaa ja korjata. Yrityksillä ja organisaatioilla tulisi olla selkeitä käytännön lähestymistapoja muistiturvallisuuden parantamiseksi. (The White House 2024b.)

Raportin mukaan kehittäjien koulutus on erityisen keskeinen rooli muistiturvallisuuden parantamisessa. Yritysten ja organisaatioiden tulisi investoida kehittäjien koulutukseen sekä muistiturvallisuuden edistämiseen keskittyviin parhaiden käytänteiden luomiseen ja ylläpitämiseen. Koulutuksen avulla kehittäjät oppivat tunnistamaan muistivirheet ja ymmärtämään niiden vaikutukset ohjelmistojen tietoturvaan. (The White House 2024b.)

Raportissa mainitaan myös yhteistyö eri tahojen välillä. Esimerkiksi hallituksen, asiantuntijoiden ja akateemisen maailman välinen yhteistyö edistää standardien kehittämistä ja tietojen jakamista muistiturvallisuuden parantamiseksi. Tämä mahdollistaa yhteisien käytäntöjen ja menetelmien kehittämisen, mistä taas seuraa parempi tietoturva koko alalla. (The White House 2024b.)

Raportissa myös korostetaan, että tietoturvauhat, myös muistiturvallisuuteen liittyvät, muuttuvat ja kehittyvät jatkuvasti, mikä tarkoittaa, että muistiturvallisuutta täytyy jatkuvasti kehittää ja innovoida. Olisi siis tärkeää, että pysyttäisiin ajan tasalla uusista uhista ja investoitaisiin ammattitaidon ylläpitoon. (The White House 2024b.)

6 MUISTITURVALLISTEN OHJELMOINTIKIELIEN HYÖDYT JA HAASTEET

Niin Valkoisen talon raportti muistiturvallisesta ohjelmistokehityksestä kuin Microsoftin ja Googlen tilastot ohjelmavirheistä ja tietoturvariskeistä korostavat muistiturvallisuuden tärkeyttä tietoturvariskien vähentämisessä. Muistiturvalliset kielet voisivat poistaa tämän riskiryhmän automaattisesti kielen muistiturvallisen rakenteen kautta. Kun taas muistiturvallisuuden saavuttaminen muistiturvattomilla ohjelmointikielillä, kuten C++-ohjelmointikielellä, vaatii ohjelmoijalta tietoa, taitoja ja huolellisuutta.

Muistinhallintavirheiden välttäminen tarkoittaa myös vakaampia ja luotettavampia ohjelmistoja (Prossimo 2024). Jos muistivirheitä ei ole, tietoturvapäivitysten ja yleisten korjausten määrä vähenee. Tämä tietysti myös säästää kustannuksia pitkällä aikavälillä. Valkoisen talon raportti korostaa erityisesti kriittisten infrastruktuurijärjestelmien muistiturvallisuutta, sillä niissä virheet voivat olla erittäin kalliita ja myös vaarallisia (The White House 2024b).

Nykyään ei välttämättä tarvitse edes tinkiä suorituskyvystä siirryttäessä muistiturvallisiin ohjelmointikieliin. Erityisesti Rust on suunniteltu matalan tason ohjelmointikieleksi, joka säilyttää esimerkiksi C++-ohjelmointikielen tunnetun korkean suorituskyvyn. (Thompson 2023.)

Muistiturvallisiin ohjelmointikieliin siirtyminen ei kuitenkaan ole niin yksinkertainen asia. Ohjelmoijat täytyy kouluttaa siirryttäessä muistiturvallisiin ohjelmointikieliin. Esimerkiksi Rustin ominaispiirteet, kuten omistajuus- ja lainaussäännöt, voivat tuntua monimutkaisilta, erityisesti siirryttäessä C++:sta tai sen kaltaisista kielistä. Kouluttaminen on suuri pitkän aikavälin sijoitus, johon yrityksillä ei välttämättä ole varaa. (Prossimo 2024.)

Vaikka esimerkiksi Rust on jatkuvasti kasvava kieli, C++-ohjelmointikielellä on pitkä historia, jonka mukana on kertynyt valtava valikoima kirjastoja ja työkaluja, joita ei välttämättä vielä löydy uudemmille muistiturvallisille ohjelmointikielille. Valkoisen talon raporttikin toteaa, ettei siirtyminen muistiturvallisiin kieliin ole helppoa (The White House 2024b). Erityisesti vanhat järjestelmät eivät välttämättä sovi uusien kirjastojen tai ohjelmistorakenteiden kanssa yhteen.

Monissa C- ja C++-ohjelmointikielillä kehitetyissä järjestelmissä on monimutkaiset rakenteet, miljoonia rivejä koodia ja erilaisia riippuvuuksia (Prossimo 2024). Tämä voi tarkoittaa, että esimerkiksi

Rustin integroiminen tällaisiin järjestelmiin olisi käytännössä mahdotonta tai vaatisi paljon uudelleenkirjoittamista. Tällaisissa tilanteissa täytyy harkita vaihtoehtoisia tapoja saavuttaa muistiturvallisempi ohjelmisto ohjelmointikielen vaihtamisen sijaan.

7 SIIRTYMINEN MUISTITURVALLISEMPAAN KOODIIN

Ohjelmistojen tietoturvaa voidaan parantaa lisäämällä muistiturvallisuutta eri tavoin. Jos tavoitteena on kokonaisvaltainen muistiturvallisuus, voitaisiin siirtyä käyttämään muistiturvallisia ohjelmointikieliä. Helpointa on käyttää muistiturvallisia ohjelmointikieliä uusissa projekteissa tai komponenteissa. Tästä seuraa, että uusi koodi on muistiturvallista eikä tulevaisuudessa tarvitse käyttää resursseja projektin uudelleen kirjoittamiseen. Jos muistiturvallisia ohjelmointikieliä käytetään uusissa projekteissa, säästytään vanhan koodin uudelleen kirjoittamiselta, ja saavutetaan silti muistiturvallisempi ohjelmisto. Tästä seuraa vähemmän tietoturvariskejä ilman suuria alkuinvestointeja. Uusien komponenttien kirjoittaminen muistiturvallisilla ohjelmointikielillä on myös mahdollista, koska useat muistiturvalliset ohjelmointikielet toimivat yhdessä muistiturvattomien C- ja C++-ohjelmointikielien kanssa. (Prossimo 2024.)

Järjestelmien kriittiset komponentit voitaisiin myös kirjoittaa uudelleen muistiturvallisella kielellä. Tämä menetelmä on hyödyllinen, jos on joitain sovelluksen tai järjestelmän osia, jotka ovat alttiimpia tietoturvariskeille ja joissa vakaus tai tietoturva ovat erityisen ratkaisevia (Prossimo 2024.). Tämä menetelmä parantaa kyseisten komponenttien tietoturvaa huomattavasti. Tämä on erityisen hyödyllinen menetelmä vanhentuneissa järjestelmissä, joissa kriittiset komponentit saataisiin suojattua ilman koko järjestelmän uusimista. Toisaalta komponenttien udelleenkirjoittaminen voi olla kallista ja haastavaa, mutta kuitenkin edullisempaa kuin koko ohjelman uudelleenkirjoittaminen.

Joissakin tilanteissa voidaan harkita koko projektin asteittaista uudelleenkirjoittamista muistiturvalliseksi. Tämä tarkoittaisi, että koko projekti korvattaisiin kokonaan täysin uudella, muistiturvallisella ohjelmointikielellä kirjoitetulla projektilla, jolloin muistivirheet ja muut turvallisuusongelmat ehkäistäisiin kielen omien muistinhallintasääntöjen kautta. Kun siirtymä on asteittainen, kielen omaksumien tapahtuu samalla. Asteittainen siirtyminen myös pienentää riskejä, sillä uusi teknologia otetaan käyttöön hallitusti. Muistiturvallisten kielten tuomat tietoturvaedut voidaan nähdä jo lyhyemmälläkin ajalla, mutta täysi muistiturvallisuus ja sen hyödyt saavutetaan vasta, kun projekti on kokonaan kirjoitettu uudestaan muistiturvallisella ohjelmointikielellä.

Koko projektin uudelleen kirjoittamisen suurina ongelmina ovat esimerkiksi korkeat kehityskustannukset ja yhteensopivuusongelmat. Tämänkaltainen siirtymä vaatii huolellista suunnittelua, doku-

mentaatiota ja asiantuntemusta molemmista ohjelmointikielistä, jotta yhteensopivuus säilyy. Esimerkiksi Philippe Gaultier kirjoittaa blogissaan kokemuksistaan uudelleenkirjoitusprosessistaan, jossa hän kirjoitti kokonaisen C++-ohjelmiston asteittain Rust-ohjelmointikielellä. Näistä hänen käytännönkokemuksistansa huomataan, että työkaluissa on vielä merkittävästi parantelun varaa ja ohjelmiston täysi uudelleenkirjoittaminen toisella ohjelmointikielellä ei ole helppoa. (Gaultier 2024.)

Yhdysvaltain asevoimien tutkimusorganisaatiolla (engl. Defense Advanced Research Projects Agency, DARPA) on käynnissä projekti, TRACTOR (Translating All C To Rust), joka automatisoisi C-ohjelmointikielen kääntämisen Rustiksi. Tavoitteena olisi saavuttaa yhtä laadukasta Rust-koodia kuin kokenut Rust ohjelmoija tuottaisi. Tämä kunnianhimoinen projekti on vielä kesken, mutta onnistuessaan helpottaisi C-ohjelmistojen uudelleenkirjoittamista muistiturvallisemmiksi. (Wallach 2024.)

Jos muistiturvallisiin ohjelmointikieliin siirtyminen ei kuitenkaan ole vaihtoehto, niin kuin se harvoin on, voidaan harkita esimerkiksi vanhempien C++-versioiden päivittämistä moderneihin versioihin (esim. C++11 ja uudemmat). Ne tarjoavat mahdollisuuden parantaa koodin muistiturvallisuutta, sillä ne sisältävät muistiturvallisuutta edistäviä ominaisuuksia, kuten älykkäät osoittimet (smart pointers), jotka automatisoivat muistinhallintaa ja vähentävät virheiden riskiä. Tämä menetelmä tarkoittaa vähemmän koodin uudelleenkirjoitusta, mutta silti saavutetaan parempi muistiturvallisuus. Moderniin C++-versioon siirtyminen on usein helppoa verrattuna koko ohjelmiston uudelleenkirjoittamiseen, mutta usein vanhaa koodia tarvitsee myös kirjoittaa uudelleen, jotta uudemmat C++-ominaisuudet saadaan käyttöön turvallisesti, mikä voi viedä aikaa ja vaatii perusteellista testaamista. (Sibony 2021.)

Jos halutaan säilyttää C++-ohjelmisto, eikä moderniin C++-versioon siirtyminen osoittaudu riittäväksi parannukseksi, vaihtoehtona voi olla, myös aiemmin mainittu, kehitteillä oleva muistiturvallisuuslaajennus C++/Safe, joka voi auttaa parantamaan muistiturvallisuutta C++-koodissa. Opinnäytetyön kirjoitusajankohtana projekti on vielä kesken, mutta onnistuessaan laajennus voi olla erinomainen työkalu muistiturvallisemman C++-koodin saavuttamiseksi. (Falco 2024.).

Voidaan myös harkita muistivirheitä tarkkailevien analyysityökalujen, kuten .NET Object Allocation tool, käyttöönottoa. Näillä työkaluilla voidaan tunnistaa ja siten myös estää muistivirheitä. Tämä lähestymistapa ei edellytä koodin uudelleenkirjoittamista, mutta silti vähentää muistivirheistä johtuvia tietoturvariskejä. Ongelmana on kuitenkin se, etteivät työkalut takaa täyttä muistiturvaa vaan

ainoastaan auttavat tunnistamaan virheitä. Lisäksi muistivirheitä voi silti jäädä analyysityökalujen ulkopuolelle. Työkalujen käyttäminen vaatii myös ylläpitoa, osaamista ja investointia. (Microsoft 2024.)

Muistiturvallisempaan koodiin siirtyminen tarkoittaa, että yritysten, organisaatioiden ja valtioiden tulisi investoida koulutukseen ja osaamisen kehittämiseen. Esimerkiksi tiimejä voitaisiin kouluttaa käyttämään muistiturvallisia ohjelmointikieliä tai ymmärtämään muistivirheiden hallintaa C++-kielellä. Koulutus parantaisi ohjelmoinnin laatua ja vähentäisi muistivirheiden riskiä projekteissa, mutta se vie aikaa ja vaatii investointia.

Yksi kannattavimmista menetelmistä on yhdistää eri menetelmiä. Harvoin yrityksissä on resursseja koko ohjelman uudelleenkirjoittamiseen. Yritysten tulisi analysoida mahdollisia vaihtoehtoja tapauskohtaisesti. Esimerkiksi yritys voisi käyttää muistiturvallista ohjelmointikieltä uusissa projekteissa, päivittää kriittisiä komponentteja asteittain ja ottaa käyttöön koodianalyysityökaluja vanhemmassa C++-koodissa. Tämä lähestymistapa mahdollistaa hallitun ja joustavan siirtymän muistiturvallisempaan ohjelmistoon ilman epärealistisen suuria riskejä ja kertakustannuksia. Tulevaisuus näyttää kuitenkin olevan siirtymässä muistiturvallisemmaksi, mikä tarkoittaa, että yritysten on kannattavaa pysyä kehityksen mukana (The White House 2024a).

8 YHTEENVETO

Opinnäytetyön aiheena oli muistiturvallisuus ja muistiturvalliset ohjelmointikielet tietoturvan näkökulmasta, muistiturvallisten ohjelmointikielten hyödyt ja haasteet sekä muistiturvallisempaan ohjelmistokehitykseen siirtyminen. Opinnäytetyön tavoitteena oli selkeyttää muistiturvallisuuden merkitystä ohjelmointikielen valinnassa sekä selvittää muistiturvallisempia vaihtoehtoja.

Opinnäytetyöstä käy ilmi, että muistiin liittyvät ohjelmavirheet ovat yleisiä, ja niistä voi seurata myös vakavia tietoturvariskejä. Tämän takia suositellaan siirtymistä muistiturvallisempaan ohjelmistokehitykseen esimerkiksi vaihtamalla muistiturvattomat ohjelmointikielet muistiturvallisiin ohjelmointikieliin, joita löytyy monipuolisesti eri käyttötarkoituksiin. Jopa Yhdysvaltain valtion tasolla suositellaan siirtymistä muistiturvallisiin ohjelmointikieliin, joten muistiturvallisuus on ajankohtainen aihe.

Muistiturvallisten ohjelmointikielten käyttöä kannattaisi harkita erityisesti tilanteissa, joissa ei ole tarvetta suoralle yhteydelle muistinkäsittelyyn, ollaan aloittamassa uutta projektia tai joissa muistivirheet voivat aiheuttaa erityisen suuria ongelmia. Suosituksista ja ehdotuksista huolimatta suurien ohjelmistojen uudelleenkirjoittaminen ei ole helppoa eikä usein taloudellisesti kannattavaa, jolloin voidaan panostaa muihin keinoihin saavuttaa muistiturvallisempi koodi.

Yksi kattavin tapa parantaa muistiturvallisuutta ilman koko ohjelmiston uudelleenkirjoittamista on ohjelmistokehittäjien kouluttaminen. Jos muistivirheet, kuten muistivuodot ja kaksoisvapautukset, osataan tunnistaa ja korjata sekä muistiturvattomien ohjelmointikielten muistinhallinnan säännöt ja parhaat käytänteet ovat selkeät, muistiturvattomien kielten kanssa voidaan työskennellä turvallisemmin. Muistiturvattomat ohjelmointikielet, kuten C ja C++, ovat vanhoja kieliä, jotka eivät tule katoamaan vuosiin, joten ammattitaitoinen muistinhallinnan osaaminen on arvokas taito.

Keskustelu muistiturvallisuudesta on kiivasta. Aiheesta vaikuttaa olevan kaksi mielipidettä eri ääripäissä. Jotkut ovat voimakkaasti koko aiheen jatkuvaa käsittelyä vastaan. Keskustelu muistiturvallisuudesta koetaan jo liian paisuneena ja todetaan, että ”huono koodi on huonoa koodia, millä kielellä vain”. Esimerkiksi Yhdysvaltain Valkoisen talon aiheesta julkaisema raportti herätti kiihkeää kritiikkiä, mutta toisaalta jotkut taas juhlistivat kyseistä raporttia. Tämä toinen ääripää näkee muistiturvalliset ohjelmointikielet ainoana ratkaisuna ja erityisesti Rustilla on suuri yhteisö, joka ylistää kielen käyttöä jokaisessa kontekstissa.

Tästä seuraa, että on yllättävän hankalaa löytää neutraalia ja ammattimaista informaatiota muistiturvallisuudesta. Jotta tekninen yhteisö voisi todellisesti kehittyä aiheen suhteen, muistiturvallisuudesta täytyisi keskustella neutraalisti ja todellisesti harkita muistiturvallisten ja muistiturvattomien ohjelmointikielien hyötyjä ja haasteita.

Muistiturvallisuus on tärkeä ja ajankohtainen aihe, josta on keskusteltu jo vuosikymmenien ajan. Mutta muistiturvallisuus on kuitenkin vain osa turvallisen koodin kirjoittamista ja vastuu tietoturvallisesta tulevaisuudesta on kaikilla teknisen yhteisön jäsenillä.

LÄHTEET

Ada 2024. Types. Ada Quality and Style Guide. Luettavissa: https://ada-lang.io/docs/style-guide/s5/03. Luettu: 6.11.2024

Anderson, B., Bergstrom, L., Herman, D., Matthews, J., McAllister, K., Goregaokar, M., Moffitt, J. & Sapin, S. 2015. Experience Report: Developing the Servo Web Browser Engine using Rust. Cornell University. Luettavissa: https://arxiv.org/abs/1505.07383. Luettu: 11.11.2024.

Asay, M. 2020. Why AWS loves Rust, and how we'd like to help. Amazon Web Services. Luettavissa: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/. Luettu: 11.11.2024

Azevedo de Amorim, A., Hriţcu, C., Pierce, B.C. 2018. The Meaning of Memory Safety. Teoksessa Bauer, L., Küsters, R. (eds) Principles of Security and Trust. Springer, Cham. Luettavissa: https://doi.org/10.1007/978-3-319-89722-6_4. Luettu: 5.11.2024

Blue Coding 2023. A Comprehensive Comparison of C# and Java for Contemporary Developers. Luettavissa: https://www.bluecoding.com/post/c-vs-java-an-in-depth-comparison-for-modern-developers. Luettu: 17.11.2024.

Codex A. 2023. A Guide to Rust's Documentation and Community Resources. Reintech. Luettavissa: https://reintech.io/blog/guide-rust-documentation-community-resources. Luettu: 6.11.2024

Deronjic V. 2023. A Comprehensive Guide to C++: Advantages and Disadvantages. Pantgea. Luettavissa: https://pangea.ai/resources/a-comprehensive-guide-to-c-advantages-and-disadvantages. Luettu: 10.11.2024

Diaz, F. 2023. How to secure memory-safe vs. manually managed languages. GitLab. Luettavissa: https://about.gitlab.com/blog/2023/03/14/memory-safe-vs-unsafe/. Luettu: 11.11.2024

Falco, V 2024. Safe C++ Partnership. C++ Alliance. Luettavissa: https://cppalliance.org/vinnie/2024/09/12/Safe-Cpp-Partnership.html. Luettu: 11.11.2024

FBI 2018. The Morris Worm. Luettavissa: https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218. Luettu: 4.11.2024

Fernando, D. 2023. How to secure memory-safe vs. manually managed languages. GitLab. Luettavissa: https://about.gitlab.com/blog/2023/03/14/memory-safe-vs-unsafe/. Luettu: 6.11.2024

Gaultier, P. 2024. Lessons learned from a successful Rust rewrite. Luettavissa: https://gaultier.github.io/blog/lessons_learned_from_a_successful_rust_rewrite.html. Luettu: 17.11.2024.

GeeksforGeeks 2022. History of C++. Luettavissa: https://www.geeksforgeeks.org/history-of-c/. Luettu: 10.11.2024

GeeksforGeeks 2024. Memory leak in C++ and How to avoid it? Luettavissa: https://www.geeksforgeeks.org/memory-leak-in-c-and-how-to-avoid-it/. Luettu: 11.11.2024

GeeksforGeeks 2024b. Dangling Pointers in C++. Luettavissa: https://www.geeksforgeeks.org/dangling-pointers-in-cpp/. Luettu: 11.11.2024

Granot, L. 2024. Memory Corruption: Examples, Impact, and 4 Ways to Prevent It. Sternum. Luettavissa: https://sternumiot.com/iot-blog/memory-corruption-examples-impact-and-4-ways-to-prevent-it/. Luettu: 5.11.2024

Gregory J. 2024. Can memory-safe programming languages kill 70% of security bugs?. Security Intelligence. Luettavissa: https://securityintelligence.com/news/memory-safe-programming-languages-security-bugs/. Luettu: 6.11.2024

Harán, J. M. 2018. Malware of the 1980s: Looking back at the Brain Virus and the Morris Worm. WeLiveSecurity. Luettavissa: https://www.welivesecurity.com/2018/11/05/malware-1980s-brain-virus-morris-worm/. Luettu: 4.11.2024

Harvard 2021. Section 2: Memory bugs. Luettavissa: https://cs61.seas.harvard.edu/site/2021/Section2/. Luettu: 11.11.2024

ISO/IEC 8652:2023. Information technology — Programming languages — Ada. Luettavissa: https://www.iso.org/standard/83621.html. Luettu: 6.11.2024

Jonna, V. 2024. Advantages And Disadvantages Of C Language (A Detailed Explanation). Ellow. Luettavissa: https://ellow.io/advantages-and-disadvantages-of-c-language/. Luettu: 11.11.2024

Kapil, D. 2020. Double Free. Heap-exploitation. Luettavissa: https://heap-exploitation.dhavalkapil.com/attacks/double_free. Luettu: 11.11.2024

Microsoft 2019a. A proactive approach to more secure code. Luettavissa: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. Luettu: 11.11.2024

Microsoft 2019b. Why Rust for safe systems programming. Luettavissa: https://msrc.microsoft.com/blog/2019/07/why-rust-for-safe-systems-programming/. Luettu: 11.11.2024

Microsoft 2019c. We need a safer systems programming language. Luettavissa: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/. Luettu: 11.11.2024

Microsoft 2021. Smart pointers (Modern C++). Luettavissa: https://learn.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170. Luettu: 11.11.2024.

Microsoft 2024. Choose a memory analysis tool in Visual Studio (C#, Visual Basic, C++, F#). Luettavissa: https://learn.microsoft.com/en-us/visualstudio/profiling/analyze-memory-usage?view=vs-2022. Luettu: 12.11.2024

Ochoa, D. 2024. Rust vs Java: Choosing the Right Tool for Your Next Project. Halfinne. Luettavissa: https://www.halfnine.com/blog/post/rust-vs-java. Luettu: 17.11.2024.

Prossimo 2024. What is memory safety and why does it matter?. Luettavissa: https://www.memorysafety.org/docs/memory-safety/. Luettu: 5.11.2024.

ReasonLabs 2023. What is Memory Leak? Luettavissa: https://cyberpedia.reasonlabs.com/EN/memory%20leak.html. Luettu: 11.11.2024.

Rebert, A. & Kern, C. 2024. Secure by Design: Google's Perspective on Memory Safety. Google. Luettavissa: https://storage.googleapis.com/gweb-research2023-media/pubtools/7665.pdf. Luettu: 11.11.2024.

Rust 2024a. Understanding Ownership. Luettavissa: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html. Luettu: 6.11.2024

Rust 2024b. What Is Ownership? Luettavissa: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. Luettu: 6.11.2024

Rust 2024c. References and Borrowing. Luettavissa: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html. Luettu: 6.11.2024

Rust 2024d. Unsafe Rust. Luettavissa: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html. Luettu: 6.11.2024

Sibony, J. 2021. How to Modernize Legacy C++ Code? Incredibuild. Luettavissa: https://www.incredibuild.com/blog/how-to-modernize-legacy-c-code. Luettu: 17.11.2024

Snyk 2024a. C++ in the wild: Which industries use C++? Luettavissa: https://snyk.io/learn/who-uses-cpp/. Luettu: 10.11.2024

Snyk 2024b. Use after free. Luettavissa: https://learn.snyk.io/lesson/use-after-free/. Luettu: 10.11.2024.

Statista 2024. Most used programming languages among developers worldwide as of 2024. Luettavissa: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. Luettu: 10.11.2024

The Chromium Projects 2024. Memory safety. Luettavissa: https://www.chromium.org/Home/chromium-security/memory-safety/. Luettu: 4.11.2024

The White House 2024a. Press Release: Future Software Should Be Memory Safe. Luettavissa: https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/. Luettu: 4.11.2024.

The White House 2024b. Back to the building blocks: a path toward secure and measurable software. Washington. Pdf-tiedosto. Luettavissa: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf. Luettu: 6.11.2024

Thompson, C. 2023. How Rust went from a side project to the world's most-loved programming language. MIT Technology Review. Luettavissa: https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/. Luettu: 6.11.2024.

University of Michigan 2024. The Ada Programming Language. Luettavissa: http://groups.umd.umich.edu/cis/course.des/cis400/ada/ada.html. Luettu: 6.11.2024

Vu, W. 2019. The Ghost of Exploits Past: A Deep Dive into the Morris Worm. Rapid7. Luettavissa: https://www.rapid7.com/blog/post/2019/01/02/the-ghost-of-exploits-past-a-deep-dive-into-the-morris-worm/. Luettu: 4.11.2024

Wallach, D. 2024. Translating All C to Rust (TRACTOR). Defense Advanced Research Projects Agency, DARPA. Luettavissa: https://www.darpa.mil/program/translating-all-c-to-rust. Luettu: 17.11.2024.