Sasu Ilmo

Ohjelmontikielen vaikutus tietoturvallisuuteen

Tietotekniikan kandidaatintutkielma

30. huhtikuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Sasu Ilmo

Yhteystiedot: sasu.ilmo@gmail.com

Ohjaaja: Jonne Itkonen

Työn nimi: Ohjelmontikielen vaikutus tietoturvallisuuteen

Title in English: The effect of programming language on cyber security

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 19+0

Tiivistelmä: Työssä tarkistellaan yleisiä tietoturvallisuusuhkia ja minkälaisia turvallisuusmekanismeja ohjelmointikielissä on niitä vastaan toteutettu. Lisäksi mainitaan jonkin verran mahdollisia turvatoimia, kuten uusia kieliä ja tapoja skannata koodia tietoturvaongelmian varalta.

Avainsanat: kybertyrvallisuus, ohjelmointikielet

Abstract: The thesis goes over common cyber security threats and what types of method programming languages have to counter them. The thesis also mentions potential fixes like new languages and methods to scan code for security threats.

Keywords: cyber security, programming languages

Jyväskylässä 30. huhtikuuta 2022

Sasu Ilmo

Sisällys

1 Johdanto

Tutkimuksen aiheena on tutkia, kuinka paljon ohjelmointikielellä on vaikutusta ohjelmien tietoturvallisuuteen. Ohjelmointikielten eri turvallisuustoimia on hyvä tutkia, sillä se auttaa ohjelmoijia valitsemaan parhaan kielen tehtävää varten lisäksi se myös auttaa ohjelmoijia ymmärtämään turvallisuusmenetelmiä kielissä, joita he jo käyttävät. Meyerovich ja Rabkin (2013) mukaan vain noin 40% ohjelmoijista pitää tärkeänä, että kieli on tietoturvallinen. Hickey (2014) mielestä kielellä ei oikeastaan ole väliä turvallisuuden suhteen. Kurilova, Potanin ja Aldrich (2014) taas pitävät kieltä erittäin tärkeänä koodin turvallisuuden kannalta. Croft ym. (2021) tutkimuksen mukaan kielissä ilmenee erilaisia tietoturvaonglemia, joka näkyy siinä, että käyttäjät eri foorumeilla kysyvät eri asioista kielestä riippuen mutta keskustelun määrässä ei kuitenkaan ole tilastotieteellisesti mitattavaa eroavaisuutta kielten välillä. Khwaja, Murtaza ja Ahmed (2020) kehittivät suojausominaisuuskehyksen (eng. security feature framework), joka on metodi jolla yritetään kvantifioida ohjelmointikielten turvallisuutta. Kun sitä sovellettiin eräisiin kieliin, niin suurin prosentti oli 68,5 % (C# ja Java) ja alin oli 42,6 % (C++). Prosentit vastaavat kuinka kattavasti kielistä löytyy turvallisuusominaisuuksia, joita tässä suojausominaisuuskehyksessä on pidetty tärkeinä. Kaikkia kieliä mitä on olemassa, ei ole mitattu tällä kehyksellä, joten voi olla mahdollista, että jotkut kielet ovat näiden rajojen ulkopuolella. Tutkielma on rakenteeltaan seuraavanlainen. Kappaleessa 2 käydään läpi yleisiä tietoturvaongelmia. Kappaleessa 3 käydään läpi mekanismeja, joita ohjelmointikielistä löytyy tietoturvan mahdollistamiseksi ja turvaamiseksi. Kappaleessa 4 käydään läpi hypoteettisia ja kielten ulkoisia ratkaisuja ongelmiin. Kappaleessa 5 on yhteenveto.

2 Yleisiä tietoturvaongelmia

Tässä kappaleessa on esitetty joitakin yleisiä tietoturvaongelmia. Lista ei ole täydellinen, mutta tämä tutkimustyö ei olisi koskaan valmis, jos lähtisin selvittämään kaikkia ongelmia, joten raja oli vedettävä johonkin. Jatkoa tutkimukselle voisi olla tutkia niitä asioita, joita jätin pois tästä tutkielmasta. Yhtenä isoimpana tietoturvariskinä voidaan pitää käyttäjän syötteen huonoa sanitointia (Adam Chaudry 2021), joka johtaa moniin eri tietoturvavikoihin esim. XSS ja SQL-injektioihin ja oikeastaan kaikkiin injektion tapaisiin tietoturvaongelmiin. Sanitoinilla tarkoitetaan sitä, että käyttäjän syöte täytyy esittää ohjelmalle sellaisessa muodossa, että sitä ei vahingossakaan toteuteta ohjelmakoodina. Isoin haavoittuvaisuus on käyttäjät itse, huonoine salasanoineen ja latailemalla netistä mitä sattuu mutta tähän aihepiiriin tämä tutkimus ei ota kantaa.

2.1 Käyttäjän syöte

Koska käyttäjä on voi oikeastaan olla kuka tahansa ja mahdollisesti tuntematon niin kaikki syöte mitä käyttäjältä saadaan pitää olettaa vaaralliseksi ja siksi sanitoitavaksi. Sanitointi on mutkikasta ja täytyy toteuttaa eri tavoilla riippuen mitä hyökkäystä vastaan ollaan puolustautumassa. Esimerkiksi jos jonkin sivun kommenttikenttää ei sanitoida niin hyökkääjä voisi vaikka kirjoittaa kommenttiinsa HTML komentosarja tunnisteen

```
<script "src=http://hyokkaajansivu.com/authstealer.js"> </script>
```

ja näin aina, kun joku käy kyseisellä sivulla niin tämä komentosarja ajettaisiin. Poll (2018) mainitsee, että on vaikea sanitoida syötettä, joka viedään eteenpäin palvelinpuolelle, koska kehittäjä ei välttämättä tiedä mitä siellä syötteelle tehdään.

2.2 Osoittimet ja muistiviat

Osoitin on muuttuja, joka sen sijaan että se sisältäisi jonkin konkreettisen arvon, kuten vaikka kokonaisluvun, se sisältää muistiosoitteen, josta löytyy jokin arvo. Esimerkiksi Haywood, Yu ja Yuan (2013) kertovat Javan sisäisistä turvallisuusmenetelmistä, kuten siitä, että ohjel-

moijat eivät itse pääse käsiksi osoittimiin, jotka tekisivät kielistä muistiturvattomia. Esimerkiksi C:ssä voidaan tehdä osoitin aritmetiikkaa eli osoittimen sisältävää muistipaikkaa voidaan muokata mielivaltaisesti johtaen osoittimen sisältämään jonkin muun muuttujan arvon tai roskaa. Roskalla tässä tarkoitetaan, että osoitin osoittaa muistiosoitteeseen, jota ei ole alustettu ja sisältää siksi mitä tahansa. Esimerkki ohjelma, jolla voidaan printataan roskaa C:ssä:

```
 #include <stdio.h>
int main()
{
    int x = 12;
        int *p = &x;
        char y = 'a';
        *p++;
        printf("x:n␣arvo␣on␣:␣%d\n",*p);
    return 0;
}
```

Ohjelman kun ajaa niin se tulee tulostamaan mitä sattuu. Okhravi (2021) analysoi Linux ohjelmia ja tuli tulokseen, että 33% haavoittuvuuksista johtui kielten muistiturvattomuudesta eli siitä, että jotakin kautta päästiin käsiksi ohjelman muistipinoon ja injektoimaan omaa koodia sinne, tai että koodissa tapahtuu yli-tai alivuotoja. Javassa taas on ongelmia syötteen sterilisoinnin kanssa, joka voi johtaa mahdollisesti SQL-injektioihin (Shahriar ja Zulkernine 2012). Dietz ym. (2015) mukaan ongelma ylivuotojen kanssa C-ja C++ kielissä on se, että usein ne ovat tarkoituksellisia. Joskus myös voi käydä siten, että aliohjelmalle annetaan osoittaja, jota ei ole alustettu eli sille ei olla annettu arvoa ja odotetaan että aliohjelma alustaa sen ja samaan aikaan aliohjelma odottaa, että sille annettu osoittaja on valmiiksi alustettu, normaalisti C:n kääntäjä luultavasti varoittaisi tästä, mutta jos nämä aliohjelmat ovat eri lähdetiedostoissa niin sitä ei kääntäjä löydä (Kelly, Gu ja Maksimovski 2021).

2.3 Vakioaikaisuus

Vakioaikaisuudella tarkoitetaan sitä, että syötteestä riippumatta koodin tai aliohjelman ajaminen kestää saman verran. Aikavaihtelulla voidaan laskea, jos väsytyshyökkäys löysi oikean merkin, vaikka salasanassa. väsytyshyökkäys toimii siten, että arvataan merkkijonoja ja yritetään saada luotua, vaikka käyttäjän salasana satunnaisesti. Tämä on normaalisti erittäin hidasta. Ajan saa kaavalla:

maksimi_aika = (mahdolliset_merkit^salasanan_pituus)/kokeilua_sekunnissa

(University of South Wales 2020). Tätä tapaa saadaan nopeutettua huomattavasti, jos voidaan laskea vain osa salasanasta kerralla, jonka aikavaihtelu mahdollistaa. Kielien sisäänrakennetuissa yhtäsuuruus metodeissa tarkistus lopetetaan heti, kun yksi kirjain merkkijonossa eroaa, vaikka kirjaimia olisi jäljellä. Brumley ja Boneh (2005) todistivat, että ajastushyökkäyksellä voidaan myös rikkoa WWW:ssä salaukseen usein käytettyä OpenSSL kirjaston toteuttama SSL-algoritmi ja pystyttiin saamaan yksityisiä avaimia. Brumley ja Tuveri (2011) Onnistuivat saamaan yksityisen avaimen OpenSSL:ää käyttävästä palvelimesta, jossa käytettiin eräänlaista ajastushyökkäyksen esto algoritmia. Nämä viat on kuitenkin jo korjattu, mutta demonstroivat kuitenkin, kuinka vakava ongelma ajastushyökkäykset ovat ja kuinka vaikea niiltä on suojautua. Toinen ongelma on, että kääntäjät voivat tehdä jonkinlaisia optimointeja, joilla koodista tulee nopeampaa, joka taas saattaa mitätöidä kehittäjän yritykset tehdä koodista vakioaikaista. Esimerkiksi Cauligi ym. (2017) tulivat tulokseen, että vaikka C:llä voidaan kirjoittaa vakioaikaista koodia, se on hankalaa, vaivalloista ja saattaa mitätöityä koska kääntäjä päätti optimoida koodia.

2.4 Määrittelemätömät toiminnot

Tässä tarkoitetaan niitä asioita, joita voidaan tehdä kielessä mutta niitä ei olla standardisoitu tai määritelty missään dokumentaatiossa. Esimerkkinä voidaan antaa C kieli, jossa ei ole määritelty kääntäjän dokumentaatiossa mitä pitää tapahtua jos jaetaan nollalla, joissakin laitteistoarkkitehtuureissa tämä aiheuttaa poikkeuksen ja toisissa se jätetään huomioimatta (Wang ym. 2012). Pahimmassa tapauksessa voi käydä niin, että myöhemmin vuosien päästä päätetään tämä standardisoida tekemään jotain muuta kuin mitä ohjelmoija oli alun perin

ajatellut, jolloin ohjelmasta tulee haavoittuvainen. On myös mahdollista, että implementaatio vaihtelee kääntäjien mukaan. Määrittelemättömiä toimintojen käyttö on erityisesti yleistä C-ja C++ kielissä, yleensä johtaen ylivuotoihin (Dietz ym. 2015). Määrittelemättömyydellä on myös etunsa. Mitä vähemmän kääntäjä puuttuu koodin sisältöön, sitä nopeampi koodia on kääntää ja ajaa. Se onko viisasta uhrata turvallisuutta nopeudella on jatkuvan väittelyn aiheena (Kelly, Gu ja Maksimovski 2021).

3 Kielten sisäiset mekanismit tietoturvalle

3.1 Osoittimet

Melkein kaikki modernit korkeatason ohjelmointikielet kieltävät ohjelmoijilta osoittimien käyttämisen ja jotkut kielet, kuten C# vaativat, että ohjelmoija ensiksi merkitsee koodissansa metodit, luokat ja muuttujat, jotka eivät ole turvallisia `unsafe` avainsanalla (Microsoft 2022), jolloin kääntäjä pitää tämän muistin erillään. `unsafe` sanalla kerrotaan kääntäjälle, että ohjelmoija itse tulee hoitamaan muistin siinä kohdassa, joten kääntäjä ei tee mitään ohjelmoijan puolesta, kuten varmista, että pysytään taulukoiden sisällä, kun sen alkioihin viitataan tai poista turhia olioita. `unsafe` avainsana antaa myös tehdä yli-ja alivuotoja. Tällä tavalla C#:sta tulee kuin C-ja C++ kielistä hyvine ja huonoine puolineen. Osoittimien estäminen poistaa monia muistivuotoja, joita roikkuvat osoittimet luovat kuten ”osottimen käyttö vapautuksen jälkeen” (eng. use after free) ja ”kahdesti vapautettu osoitin” (eng. double free). Tämäntapaiset haavoittuvaisuudet näyttäisivät olleen suosittuja ainakin vuosien 2008 ja 2011 välillä (Caballero ym. 2012), uudempaa dataa en ole löytänyt. Osoittimen käyttö vapautuksen jälkeen voidaan estää antamalla osoittimelle arvoksi `NULL`. Osoittimien poistaminen tietysti rajoittaa kehittäjiä (Microsoft 2022). Microsoft (2022) C# dokumentaation mukaan osoittimia tarvitaan esimerkiksi, kun halutaan toimia käyttöjärjestelmän kanssa tai joissakin aikakriittisissä algoritmeissä, joita on hankala tai mahdoton toteuttaa muilla tavoilla.

3.2 Sanitointi

Sanitointi mahdollistetaan monissa kielissä käyttämällä jäsenneltyä esitystä ja erikoisnotaatioilla, joilla voidaan turvallisesti liittää merkkijonoja ja ohjelman muuttujia tai käyttäjän syötettä. Kurilova ym. (2014) pitävät tässä tavassa ongelmana sitä, että usein nämä yksiselitteiset erikoisnotaatiot eivät välttämättä toimi keskenään yksiselitteisesti ja niistä on vaikea selvittää mahdollisia haavoittuvaisuuksia. Esimerkki C#:ssa ei turvallisesta SQL pyynnöstä:

```
"SELECT Count(∗) FROM Users WHERE UserName=’ " + user +
"’ AND Password=’" + pass + "’"
```

jos tässä toteutuksessa käyttäjä antaa user kenttään nimeksi `' Or 1=1 –` niin palvelin laskisi kaikki rivit `users` listasta. Alla on toteutettu C# niin kutsuttu valmisteltu kysely (eng. prepared statement), jolla SQL kyselyt voidaan parametrisoida ja täten sanitoida.

```
private void SQLparametritetty(int id, string desc)
{
string palvelin = "server=localhost;database=northwind;uid=sa;pwd=;";

SqlConnection yhteys = new SqlConnection(palvelin);
SqlCommand command = new SqlCommand(null, yhteys);

command.CommandText =
    "INSERT INTO Region (RegionID, RegionDescription) " +
    "VALUES (@id, @desc)";
SqlParameter idParam = new SqlParameter("@id", SqlDbType.Int, 0);
SqlParameter descParam =
new SqlParameter("@desc", SqlDbType.Text, 100);
idParam.Value = id;
descParam.Value = desc;
command.Parameters.Add(idParam);
command.Parameters.Add(descParam);
}
```

Yllä olevassa koodissa käyttäjän syöte muutetaan C# koodin mukaisiksi muuttujiksi, joita voidaan käsitellä kuten muitakin normaaleja muuttujia ja kun lopuksi nämä kasataan SQL kyselyksi niin ei teoriassa pitäisi olla vaaraa injektiosta. Tämäkään ei kuitenkaan ole täydellinen, sillä dynaamisista SQL kyselyistä on mahdoton tehdä täysin turvallisia koska joku voisi mahdollisesti keksiä tavan luoda injektion tätä parametritusta vastaan ja todellisesti turvallinen toteutus vaatisi, että kyselyt luotaisiin proseduurina SQL palvelimella (Litwin 2019). Proseduureja ei kuitenkaan tietääkseni kaikki SQL palvelimet tue, joten jos on pakko tehdä dynaamisia kyselyitä niin kannattaa ainakin tarkistaa pystyykö niistä tekemään parametrisiä tai tarkistaa löytyykö kielestä jonkinlainen erikoisnotaatio jolla sanitoida kyselyn.

3.3 Vakioaikaisuus

Monissa kielissä ei ole valmiiksi toteutettuja algoritmeja vakioaikaisuudelle ja tämä joudutaan joko toteuttamaan itse tai lataamaan WWW:stä jonkun kolmannen osapuolen toteuttamana. Kumpikaan näistä vaihtoehdoista ei ole ideaalinen. Tietoturvallisuudessa on pitkään pätenyt viisaus, että ei kannata koskaan itse toteuttaa mitään kryptografisia algoritmeja, koska usein niistä löytyy paremmin toteutettu versio jo kielen kirjastoista tai WWW:stä. WWW:stä lataamisen kanssa täytyy kuitenkin olla tarkkana ja käyttää vahvaa lähdekritiikkiä, ettei avaa ohjelmaa muille haavoittuvaisuuksille, joita saattaa olla piilossa ohjelmakoodissa. WWW:stä ladatessa kirjastoja on myös hyvä olla tietoinen riippuvuusketjuhyökkäyksistä. Riippuvuusketjuhyökkäyksillä tarkoitetaan hyökkäystä, jossa pyritään aiheuttamaan jonkin lainen haavoittuvaisuus ohjelmaan saamalla ,joko itse ohjelmaan tai johonkin sen käyttämään kirjastoon hyökkääjän kirjoittamaa koodia. Esimerkiksi jos vaikka jokin kirjasto on avoimenlähteen projekti niin joku hyökkääjä voisi mahdollisesti saada omaa pahaa koodia projektiin mukaan, jonka jälkeen kaikki ohjelmat, jotka käyttävät tätä uutta päivitettyä kirjastoa olisivat alttiita tälle haavoittuvuudelle. Lisäksi kuten aikaisemmin mainitsin Brumley ja Boneh (2005) ja Brumley ja Tuveri (2011) löysivät haavoittuvuuksia erittäin suosituista kryptografisesta kirjastosta OpenSSL, eli ei kannata luottaa sokeasti WWW:stä löytyviin projekteihin.

3.4 Määrittelemättömät toiminnot

Määrittelemättömistä toiminnoista voidaan päästä eroon ihan vain sillä, että ne standardisoidaan ja kääntäjille määrätään tietyt toiminnot niiden kohdalla. Ongelma on siinä, että monesti ne ovat tarkoituksella määrittelemättömiä, jotta kehittäjillä olisi enemmän vapautta kääntäjien toteutuksessa (Wang ym. 2012).

3.5 Ohjelmointi kielten tyypitys

Ohjelmointi kielet voivat olla staattisia, dynaamiseia tai siltä väliltä. Staattisissa kielissä muuttujien tyypit tarkistetaan käännön aikana esimerkiksi C, C++ ja Java ovat staattisia kieliä. C# on myös staattinen kieli, mutta siinä voidaan määritellä oliot erikseen dynaamisiksi

`dynamic` avainsanalla jolloin nämä oliot arvioidaan vasta ajon aikana. Virallisesti olio pysyy staattisena, mutta kääntäjä olettaa, että muuttujalle kaikki operaatiot ovat sallittuja (Microsoft 2021). Dynaamisissa kielissä muuttujien tyypit tarkistetaan vasta ajon aikana, esimerkiksi JavaScript ja Python ovat dynaamisesti tyypitettyjä kieliä. Dynaamiset kielet ovat suositumpia kuin Staattiset kielet (Meyerovich ja Rabkin 2013). Syitä tälle eivät vastaajat antaneet mutta datasta saatiin selville, että 33% piti staattista tyypitystä tärkeänä turvallisuudelle ja vain 8% piti sitä tärkeänä virheiden löydössä. Tutkijat kuitenkin huomauttavat, että prosentteja voi vääristää se, että kyselyyn vastanneet henkilöt ovat web-koodaajia, jossa suositaan dynaamisia kieliä. Esimerkkinä siitä miten virheitä voidaan löytää staattisella tyypityksellä esitetään Oraclen Groovyn dokumentaatiossa Oracle (2015) näin:

number = 5
numbr = (number + 15) / 2 // note the typo

Yllä olevassa esimerkissä dynaamisesti kirjoitettu kieli ei huomaa mitään vikaa, vaikka kehittäjällä olisi tapahtunut kirjoitusvirhe muuttujan kanssa ja myöhemmin jos koodissa käytetään number muuttujaa niin se tulee olemaan 5 eikä 10 jonka sen haluttaisiin olevan. Myös Schumacher (2015) on samaa mieltä sen kanssa, että staattisten kielten suurin etu on löytää virheitä ennen kuin ne pääsevät valmiiseen ohjelmaan. Dynaamisissakin kielissä on kuitenkin vaihtelua sen suhteen mitä ne sallivat ohjelmoijien tehdä tyyppien kanssa. Esimerkkinä voidaan käyttää JavaScriptiä ja Pythonia. Alla oleva koodi ei kääntyisi Pythonissa sillä siinä ei ole sallittua kahden eri tyypin pluslaskut ajon aikana mutta kääntyy JavaScriptissä sillä siinä tämä operaatio on sallittu.

numero = 1
kirjain =’1’
numero = numero + kirjain

JavaScriptissä muuttujasta numero, joka oli aluksi numero muuttuu merkkijono, jonka arvo on “11". Tyypitys sitoutuu myös vahvasti muisti turvallisuuteen, sillä vaikka C on staattisesti tyypitetty kieli se sallii osoitinaritmetiikan, jolla voidaan saada väärän tyyppisiä arvoja muuttujiin kuten aiemmassa osiossa on esitetty.

4 Hypoteettiset ja teoreettiset ratkaisut

4.1 Kielten ulkoiset muutokset

Okhravi (2021) tuli tulokseen, että muistiviat ovat kilpajuoksua ohjelmoijien ja hyökkääjien välillä. Ratkaisu vaatisi muutosta käyttöjärjestelmä- ja suoritinarkkitehtuuriin. Käytännössä nämä muutokset vaatisivat, että tieto, jota suorittimet käsittelisivät, sisältäisivät metatietoa siitä mitä suoritin käsittelee, jotta voitaisiin yrittää estää muistivuodot.

4.2 Uudet kielet

Cauligi ym. (2017) puhuvat kokonaan uuden kielen kehittämisestä, jossa ohjelmoijat pystyisivät merkitsemään mitkä osat ovat tärkeitä kyberturvallisuudelle, jotta kääntäjä ei tekisi muutoksia niissä kohdissa koodia. Kurilova, Potanin ja Aldrich 2014 kertovat kanssa uudesta kielestä joka on jo toteutettu, joka on suunniteltu välttämään OWASP:in määrittelemät kymmenen pahinta haavoittuvaisuutta web-ohjelmissa OWASP Foundation (2021). Suurimpana etuna pidetään tässä kielessä sitä, että siihen voidaan upottaa mikä tahansa muu kieli ja näin tehdä esimerkiksi SQL-kyselyistä yhtä helppoja kuin merkkijonojen kirjoittamisesta. He pitävät juuri tärkeimpänä turvallisuusmekanismina sitä, kuinka helppoa turvallista koodia on kirjoittaa, sillä se heidän mukaansa kehittäjät menevät usein siitä, mistä aita on matalin ja täten yleensä kirjoittavat haavoittuvaista koodia. Toinen uusi kieli GSL (Gradual Security Lannguage), jonka kehittivät Toro, Garcia ja Tanter (2018), jonka ideana on muuttaa kielten tyypitys siten, että niissä olisi turvatasot. Esimerkkinä he antavat kuinka kokonaisluku muuttujalle voitaisiin antaa korkea turvataso ja, jos sitä halutaan käyttää niin aliohjelmilla ja metodeilla täytyy olla myös saman tai korkeamman turvatason oikeudet.

4.3 Uudet menetelmät skannata koodia

Tutkiessa eri tietoturvaongelmia löysin monia erilaisia toteutuksia ohjelmille jotka käyvät koodia läpi ja tutkivat jos siitä löytyy haavoittuvaisuuksia. (Reparaz, Balasch ja Verbauwhede 2017), (Dhurjati ja Adve 2006),(Dietz ym. 2015), (Caballero ym. 2012). Tämä on hyvä tapa

löytää ongelmia ennen kuin ne joutuvat valmiiseen ohjelmaan, mutta useimmat näistä ovat yhden ongelmatyypin löytäviä ratkaisuja ja vaikka moni väittää, että heidän toteutuksensa on nopea, niin en kuitenkaan ole varma miten käytännöllistä se olisi, jos näitä ajaisi, vaikka monta kymmentä aina kun etsii koodista tietoturvahaavoittuvaisuuksia. Tietysti jos nämä tekniikat voitaisiin yhdistää yhdeksi isoksi ohjelmaksi, se tietysti tekisi näistä työkaluista käytännöllisempiä. Lisäksi moni näistä toimii vain yhdessä kielessä.

5 Yhteenveto

Kielet ovat erikoistuneet eri asioihin, joten niistä saattaa puuttua suojauksia tiettyjä haavoittuvaisuuksia vastaan tai ne eivät riittäviä, harvemmin kukaan tekee nettipalvelimia Fortranilla. On siis tärkeätä valita sopiva kieli tehtävää varten. Yleisesti kuitenkin kieliin löytyy kirjastoja, joilla näitä vajaavaisuuksia voidaan paikata, niiden kanssa kuitenkin täytyy olla tarkkana, koska jos ei ymmärrä mitä käyttää niin voi käydä huonosti ja pahimmassa tapauksessa tekee ohjelmasta haavoittuvaisen jollain muulla tavalla. Korkean tason kielet ovat yleisesti turvallisempia kuin matalan tason, johtuen siitä, että matalan tason kielet ovat lähellä tietokoneen osien-ja käyttöjärjestelmän rajapintaa ja ovat rajoitettuja vain laitteiston ja käyttöjärjestelmän rajoitusten mukaan. Vanhoissa kielissä, kuten C:ssä on enemmän myös vapauksia toteuttaa koodia, koska ne ovat suunniteltu imitoimaan matalia kieliä ja siksi niiden kääntäjissä ja standardeissa on harvemmin estoja huonolle tai haavoittuvaiselle koodille kuten muistivuodoille ja osoittimille. Lisäksi tietoturva on kasvanut tutkimus aiheena laajasti 2000-luvulla, joten monia ongelmia ei ollut edes löydetty tai tajuttu kun näitä kieliä suunniteltiin. Monessa ehdotuksessa puhutaan uusista kielistä, jotka olisivat suunniteltu tietoturva mielessä alusta asti. Lisätutkimuksen aiheeksi jää korjaisiko tämä ongelmia vai tekisikö se vain uusia vanhojen tilalle. Lisäksi voisi olla hyvä tutkia miten nämä useat koodin tarkistimet toimivat yhdessä ja kuinka mahdollista olisi näiden yhdistäminen yhdeksi isoksi ohjelmaksi.

Lähteet

Adam Chaudry, Kerry Crouse, Steve Christey Coley. 2021. *2021 CWE Top 25 Most Dangerous Software Weaknesses.* https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

Brumley, Billy Bob, ja Nicola Tuveri. 2011. “"Remote Timing Attacks Are Still Practical"”. Teoksessa *"Computer Security – ESORICS 2011",* toimittanut Vijay "Atluri ja Claudia" Diaz, "355–371". "Berlin, Heidelberg": "Springer Berlin Heidelberg". ISBN: "978-3-642-23822-2".

Brumley, David, ja Dan Boneh. 2005. “Remote timing attacks are practical”. Web Security, *Computer Networks* 48 (5): 701–716. ISSN: 1389-1286. https://doi.org/https://doi.org/10.1016/j.comnet.2005.01.010. https://www.sciencedirect.com/science/article/pii/S1389128605000125.

Caballero, Juan, Gustavo Grieco, Mark Marron ja Antonio Nappa. 2012. “Undangle: Early Detection of Dangling Pointers in Use-after-Free and Double-Free Vulnerabilities”. Teoksessa *Proceedings of the 2012 International Symposium on Software Testing and Analysis,* 133–143. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery. ISBN: 9781450314541. https://doi.org/10.1145/2338965.2336769. https://doi-org.ezproxy.jyu.fi/10.1145/2338965.2336769.

Cauligi, Sunjay, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala ja Deian Stefan. 2017. “FaCT: A Flexible, Constant-Time Programming Language”. Teoksessa *2017 IEEE Cybersecurity Development (SecDev),* 69–76. https://doi.org/10.1109/SecDev.2017.24.

Croft, Roland, Yongzheng Xie, Mansooreh Zahedi, M. Ali Babar ja Christoph Treude. 2021. “An empirical study of developers’ discussions about security challenges of different programming languages”. *Empirical Software Engineering* 27, numero 1 (joulukuu): 27. ISSN: 1573-7616. https://doi.org/10.1007/s10664-021-10054-w. https://doi.org/10.1007/s10664-021-10054-w.

Dhurjati, D., ja V. Adve. 2006. “Efficiently Detecting All Dangling Pointer Uses in Production Servers”. Teoksessa *International Conference on Dependable Systems and Networks (DSN’06),* 269–280. https://doi.org/10.1109/DSN.2006.31.

Dietz, Will, Peng Li, John Regehr ja Vikram Adve. 2015. “Understanding Integer Overflow in C/C++”. (New York, NY, USA) 25, numero 1 (joulukuu). ISSN: 1049-331X. https://doi.org/10.1145/2743019. https://doi-org.ezproxy.jyu.fi/10.1145/2743019.

Haywood, Adley, Huiming Yu ja Xiaohong Yuan. 2013. “Teaching Java security to enhance cybersecurity education”. Teoksessa *2013 Proceedings of IEEE Southeastcon,* 1–6. https://doi.org/10.1109/SECON.2013.6567447.

Hickey, Kathleen. 2014. “Most secure Web programming language? It depends” (maaliskuu). https://gcn.com/cybersecurity/2014/04/most-secure-web-programming-language-it-depends/297180/.

Kelly, Terence, Weiwei Gu ja Vladimir Maksimovski. 2021. “SchröDinger’s Code: Undefined Behavior in Theory and Practice”. *Queue* (New York, NY, USA) 19, numero 2 (huhtikuu): 28–44. ISSN: 1542-7730. https://doi.org/10.1145/3466132.3468263. https://doi.org/10.1145/3466132.3468263.

Khwaja, Amir A., Muniba Murtaza ja Hafiz F. Ahmed. 2020. “A security feature framework for programming languages to minimize application layer vulnerabilities”. *SECURITY AND PRIVACY* 3 (1): e95. https://doi.org/https://doi-org.ezproxy.jyu.fi/10.1002/spy2.95. eprint: https://onlinelibrary-wiley-com.ezproxy.jyu.fi/doi/pdf/10.1002/spy2.95. https://onlinelibrary-wiley-com.ezproxy.jyu.fi/doi/abs/10.1002/spy2.95.

Kurilova, Darya, Cyrus Omar, Ligia Nistor, Benjamin Chung, Alex Potanin ja Jonathan Aldrich. 2014. “Type-Specific Languages to Fight Injection Attacks”. Teoksessa *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security.* HotSoS ’14. Raleigh, North Carolina, USA: Association for Computing Machinery. ISBN: 9781450329071. https://doi.org/10.1145/2600176.2600194. https://doi-org.ezproxy.jyu.fi/10.1145/2600176.2600194.

Kurilova, Darya, Alex Potanin ja Jonathan Aldrich. 2014. “Wyvern: Impacting Software Security via Programming Language Design”. Teoksessa *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools,* 57–58. PLATEAU ’14. Portland, Oregon, USA: Association for Computing Machinery. ISBN: 9781450322775. https://doi.org/10.1145/2688204.2688216. https://doi-org.ezproxy.jyu.fi/10.1145/2688204.2688216.

Litwin, Paul. 2019. *Data Security: Stop SQL Injection Attacks Before They Stop You.* https://docs.microsoft.com/en-us/archive/msdn-magazine/2004/september/data-security-stop-sql-injection-attacks-before-they-stop-you.

Meyerovich, Leo A., ja Ariel S. Rabkin. 2013. “Empirical Analysis of Programming Language Adoption”. *SIGPLAN Not.* (New York, NY, USA) 48, numero 10 (lokakuu): 1–18. ISSN: 0362-1340. https://doi.org/10.1145/2544173.2509515. https://doi-org.ezproxy.jyu.fi/10.1145/2544173.2509515.

Microsoft. 2021. *Using type dynamic Csharp Programming Guide.* https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/using-type-dynamic.

———. 2022. *Unsafe code.* https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/unsafe-code.

Okhravi, Hamed. 2021. “A Cybersecurity Moonshot”. *IEEE Security Privacy* 19 (3): 8–16. https://doi.org/10.1109/MSEC.2021.3059438.

Oracle. 2015. https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html.

OWASP Foundation. 2021. https://owasp.org/www-project-top-ten/.

Poll, Erik. 2018. “LangSec Revisited: Input Security Flaws of the Second Kind”. Teoksessa *2018 IEEE Security and Privacy Workshops (SPW),* 329–334. https://doi.org/10.1109/SPW.2018.00051.

Reparaz, Oscar, Josep Balasch ja Ingrid Verbauwhede. 2017. “Dude, is my code constant time?” Teoksessa *Design, Automation Test in Europe Conference Exhibition (DATE), 2017,* 1697–1702. https://doi.org/10.23919/DATE.2017.7927267.

Schumacher, Toni. 2015. “Static vs. Dynamic Typing”. *Concepts of Programming Languages–CoPL’15* 22.

Shahriar, Hossain, ja Mohammad Zulkernine. 2012. “Mitigating Program Security Vulnerabilities: Approaches and Challenges”. *ACM Comput. Surv.* (New York, NY, USA) 44, numero 3 (kesäkuu). ISSN: 0360-0300. https://doi.org/10.1145/2187671.2187673. https://doi-org.ezproxy.jyu.fi/10.1145/2187671.2187673.

Toro, Matías, Ronald Garcia ja Éric Tanter. 2018. “Type-Driven Gradual Security with References”. *ACM Trans. Program. Lang. Syst.* (New York, NY, USA) 40, numero 4 (joulukuu). ISSN: 0164-0925. https://doi.org/10.1145/3229061. https://doi.org/10.1145/3229061.

University of South Wales. 2020. *Brute Force Password Hacking: How long will it take to Brute Force a password.* https://uwnthesis.wordpress.com/2020/07/01/brute-force-password-how-long-will-it-take-to-brute-force-a-password/.

Wang, Xi, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich ja M. Frans Kaashoek. 2012. “Undefined Behavior: What Happened to My Code?” Teoksessa *Proceedings of the Asia-Pacific Workshop on Systems.* APSYS ’12. Seoul, Republic of Korea: Association for Computing Machinery. ISBN: 9781450316699. https://doi.org/10.1145/2349896.2349905. https://doi-org.ezproxy.jyu.fi/10.1145/2349896.2349905.