Matti Lehtoniemi

WWW-sovellusten loogiset haavoittuvuudet

Tietotekniikan kandidaatintutkielma

5. tammikuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Matti Lehtoniemi

Yhteystiedot: matti.lehtoniemi@student.jyu.fi

Ohjaaja: Jonne Itkonen

Työn nimi: WWW-sovellusten loogiset haavoittuvuudet

Title in English: Logical vulnerabilities in web applications

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 24+0

Tiivistelmä: Tutkielman tavoitteena on luoda selkeä kuva siitä, mitä WWW-sovellusten loogiset haavoittuvuudet ovat teoreettisesti, miten ne ovat realisoituneet ja mitä niiden torjumiseksi voidaan tehdä. Taustalla on tarve ymmärryksen parantamiseksi miten haavoittuvuus voi syntyä ja mihin se voi johtaa yhä laajemmin käytössä olevissa WWW-sovelluksissa.

Tyypiltään tutkielma on kirjallisuuskatsaus. Tieto on kerätty aiemmista tutkimuksista ja laatu pyritty varmistamaan vertaisarvioimalla niiden sisältöä, tarkastelemalla julkaisupaikkaa ja siteerauksien määrää.

Avainsanat: Haavoittuvuus, WWW-sovellus, Looginen haavoittuvuus

Abstract: The purpose of this theses is to create a clear picture of what logical vulnerabilities are in web applications, how they have been encountered and what can be done to combat them. Reseach on the subject isn't comprehensive and there is a need to further increase the body of knowledge on what a vulnerability of this type is capable of.

The theses is of the literature type, where information has been collected from previous works. The quality of which has been insured by reviewing what the work contains, the place of publishing as well as the amount of citations.

Keywords: Vulnerability, Web application, Logical vulnerability

Termiluettelo

Sovelluslogiikka	Vaatimusten ja suunnittelun toteutus sovelluksessa (Boyer ja Moore 1979)
Looginen haavoittuvuus	Sovelluskoodissa olevan logiikan virheellisestä toteutuksesta johtuva haavoittuvuus (Aslam, Krsul ja Spafford 1996)
Injektio	Haavoittuvuus joka antaa hyökkääjälle mahdollisuuden muokata SQL/XML kutsun rakennetta (Anley 2002)
Sumea logiikka	Lähestymistapa laskentaan, joka perustuu totuusasteisiin tavanomaisen tosi tai epätosi järjestelmän sijaan (Zadeh 1988)
WWW-Sovellus	ohjelmistosovellus, jonka suoritus riippuu verkosta (Gellersen ja Gaedke 1999)
Hakurobotti	tietokantaohjelma, joka hakee verkkosivuja ja indeksoi niistä tiedonhakua varten tiettyjä kenttiä (Eichmann 1994)

Kuviot

Kuvio 1. Teoreettinen esimerkki WWW-sovelluksen loogisesta haavoittuvuudesta 6
Kuvio 2. Esimerkki reaalimaailman loogisesta haavoittuvuudesta (Pellegrino ja Balzarotti 2014) 9

Sisällys

1 JOHDANTO 1

2 TAUSTATIEDOT 3
2.1 WWW-sovellus 3
2.2 Haavoittuvuus 4

3 LOOGINEN HAAVOITTUVUUS 5
3.1 Yleinen määritelmä 5
3.2 Tarkempi määritelmä 7
3.3 Reaalimaailman esimerkkejä 8

4 EHKÄISEMINEN 10
4.1 Automaattisen havaitsemisen ongelma 10
4.2 Menettelytapoja automaatioon 11
4.3 Kehittäjän vastuu 13

5 YHTEENVETO 15

LÄHTEET 16

1 Johdanto

Yhä useammat palvelut kaupankäynnistä pankkeihin ovat WWW-sovelluspohjaisia, eli niitä käytetään suoraan verkkoselaimen kautta. Vaikka tämä tarjoaa hyötyä käytön helppoudesta datan käsittelyyn, on kumminkin huomioitava, että uuden kehityksen myötä syntyy myös uusia tapoja väärinkäyttää järjestelmiä. Selainpohjainen sovellus on helposti kaikkien käyttäjien saatavilla, joka tekee niistä myös hyökkääjien ensisijaisia kohteita (Pellegrino ja Balzarotti 2014).

Nabi, Yong ja Tao (2019) huomioivat tutkimuksessaan erityispiirteitä verkkoon keskittyvässä sovelluskehityksessä, joka ilmenee uusien palveluiden tarjontana, nopean kehittämisenä ja siirtymisenä pois perinteisistä sovelluksista täysin WWW-pohjaisiin. Kehittämisprosessi on tehokkaammin tuettu ja suorituskyky sekä käyttäjäkokemus parempia. Tämän myötä sovellusten monimutkaisuus sekä koko ovat kumminkin kasvaneet huomattavasti. Skaalan suurentuessa kehittäjä voi tehdä aiempaa helpommin virheen ja siksi on tärkeää erottaa WWW-sovellusten haavoitustyyppejä toisistaan, sekä ymmärtää miten ne toimivat ja miten niitä voidaan välttää tulevaisuudessa. Tutkielmassa keskitytään yhteen näistä tyypeistä, loogiseen haavoittuvuuteen, koska niihin liittyvä tutkimus on suppeaa, mutta ymmärrys on aiempaa tärkeämpää sovellusten kehittyessä suuremmiksi.

Looginen haavoittuvuus on pohjimmillaan virhe sovelluksen suunnittelussa, eikä sitä yleensä voida korjata helposti muokkaamalla osaa komponenttien koodista. Ongelma on monimutkainen ja korjaavana toimenpiteenä on usein oltava virheellisen komponentin uudelleensuunnittelu ja toteutus (Deepa ja Thilagam 2016). Korjaus kuluttaa aikaa ja pääomaa, mutta toimenpiteitä on pakko tehdä, sillä korjaamaton virhe johtaa ominaisuuteen joka aiheuttaa odottamattoman tuloksen. Virheellinen sovelluksen eteneminen voi johtaa laajaan määrään haittavaikutuksia, ja koska virhe on suunnittelussa, ei loogiselle haavoittuvuudelle ole tiettyä helposti tunnistettavaa esiintymistapaa. Esimerkiksi verrattuna helpommin tunnistettavaan injektioon, looginen haavoittuvuus on täysin riippuvainen sovelluksen toimintalogiikasta.

Rakenteeltaan tutkielma koostuu taustatiedoista, loogisen haavoittuvuuden määritelmästä, reaalimaailman esimerkeistä, millä tavoin haavoittuvuuksia voidaan välttää tulevaisuudessa

ja mihin tutkimus keskittyy. Tavoitteena on helposti luettava ja ymmärrettävä kirjallisuuskatsaus, josta on hyötyä haavoittuvuuksien määrittelyssä. Tuloksen laatu pyritään turvaamaan vertaisarvioimalla lähteiden sisältöä, jotka valitaan luotettavuuden perusteella julkaisupaikan ja siteerauksien mukaan sisällön arvioinnin lisäksi.

2 Taustatiedot

Ennen loogisista haavoittuvuuksista puhumista katsellaan taustatietoa, joka on tarpeellista myöhemmin käytettävien käsitteiden ymmärtämiseksi. Luvuissa 2.1 ja 2.2 käsitellään WWW-sovelluksia ja haavoittuvuuksista yleisesti, koska aiheet ovat tiukasti sidoksissa loogisiin haavoittuvuuksiin.

2.1 WWW-sovellus

WWW-sovelluksella tarkoitetaan ohjelmaa joka toimii internetissä ja jolla on tietty hyödyllinen tavoite. Käyttäjä tarvitsee vain aktiivisen verkkoyhteyden, jotta pääsee käsiksi palveluun. Sovelluksen arkkitehtuuriin kuuluva palvelin hoitaa tiedon hakemisen ja tallentamisen, joka esitetään sitten asiakkaalle verkkoselaimen kautta. Sivustot ovat sekoitus HTML:llä ilmaistua sisältöä, esittämisohjeita ja skriptejä (Conallen 1999). Näin käyttäjä voi olla vuorovaikutuksessa esimerkiksi lomakkeiden, tietokantojen tai ostoskorien kanssa.

Etuna perinteiseen sovellukseen on laaja skaalautuvuus ja monipuolisuus. Sovellukset voivat vaihdella pienestä ja lyhytikäisestä palveluista laajoihin monille palvelimille jaettuihin järjestelmiin joilla on miljoonia käyttäjiä (Gellersen ja Gaedke 1999). Yksi syy tähän on webselainten laaja levinneisyys. Keskittämällä prosessointi sovelluksen palvelimelle, voidaan mahdollistaa kuluttajalle universaali sekä matalan kynnyksen pääsy sovellukseen. Käyttäjä ei itse tarvitse suurta prosessointitehoa ja käyttäjäkokemus on parempi, koska sovellusta ei tarvitse asentaa, eikä resursseja kulu yhteensopivuusongelmien ratkaisemiseen. Tällaisten niin sanottujen ohutasiakkaiden lisäksi verkkopalveluiden etuna on myös keskitetty ylläpito, mikä helpottaa laskemaan kustannuksia ohjelmistopäivitysten käyttöönotossa (Gellersen ja Gaedke 1999). Esimerkiksi *Software as a Service* (lyh. **SaaS**) palvelut, eli palvelut jotka tarjoavat sovelluksen käyttöoikeuden tilauksesta, voivat varmistaa kaikkien käyttäjien käyttävän samaa versiota.

Vaikka WWW-sovellukset tarjoavat etuja, on haavoittuvuuksien kannalta merkittävää, että tietoturva-aukko voi johtaa perinteistä sovellusta laajempaan ja haitallisempaan lopputulokseen. Tiedon tallentaminen verkkoon tekee palveluista haavoittuvaisempia (Apostu ym. 2013),

koska mahdollisuus tiedon vuotamiseen kasvaa. Jos yksi käyttäjä löytää haavoittuvuuden, pystyy hän mahdollisesti vaikuttamaan kaikkiin muihin palvelua käyttäviin asiakkaisiin.

2.2 Haavoittuvuus

Määrittelyjä haavoittuvuudelle monia ja objektiivinen menetelmien yleistäminen on siksi vaikeaa. Alkaen Linde (1975) julkaisusta ohjelmistotutkijat ovat kehittäneet monia malleja turvallisten sovellusten kehittämiseen ja haavoittuvuuksien analyysiin. Tutkielmassa käytetään useaa määritelmää yhdessä, koska ne tarjoavat helposti ymmärrettävän selityksen, joka sopii myös loogisten haavoittuvuuksien määrittelyyn.

Deepa ja Thilagam (2016) määritelmän mukaan haavoittuvuus tarkoittaa tietoturva-aukkoa tai virhettä sovelluksessa, joka syntyy ohjelmointi- tai suunnitteluvirheestä ja jonka myötä on mahdollista väärinkäyttää sovellusta ja aiheuttaa vahinkoa. Amoroso (1994) tarkentaa, että uhka voi olla mikä tahansa tapahtuma, on se pahatahtoinen tai ei, kunhan se vaikuttaa tietokonejärjestelmän resursseihin ja sen myötä toimintaan. Nabi, Yong ja Tao (2021) taas jakavat haavoittuvuuksien määrittelyn karkeasti kahteen luokkaan, jotka ovat tekniset ja loogiset haavoittuvuudet, jotka kummatkin aiheuttavat jonkinlaisia haittavaikutuksia. Negatiivinen vaikutus voi vaihdella pienestä ärsykkeestä katastrofiseen efektiin, joka vaikuttaa toimeentuloon ja jopa elämään. Esimerkiksi haitallinen koodi-injektio voi johtaa tietovuotoon tai suunnitteluvirhe lääkinnällisessä laitteessa ihmisen menehtymiseen (Leveson ym. 1995).

Jotta tietokonejärjestelmät pystyisivät toimimaan turvallisesti, pyritään olemassa olevien haavoittuvuuksien hyväksikäyttö estää järjestelmillä kuten palomuureilla. Tilanteissa, joissa ei ole mahdollista eliminoida hyväksikäytön riskiä pyritään havaitsemaan väärinkäyttö automaattisten työkalujen avulla. Yleinen tapa tunnistaa virheitä, jotka johtavat tietoturvaongelmiin erityisesti WWW-sovelluksissa on käyttää automaattisia haavoittuvuusskannereita (Bau ym. 2010). Näillä pyritään tunnistamaan haavoittuvuuksia ennen niiden ilmenemistä sovelluksen ollessa käytössä.

3 Looginen haavoittuvuus

Loogisen haavoittuvuuden määrittelemiseksi kappaleissa 3.1 ja 3.2 tarkastellaan eri näkökulmista miten haavoittuvuustyyppi voi ilmetä yleisesti ja mitä tarkemmat määritelmät ovat. Kappaleessa 3.3 käsitellään reaalimaailman esimerkkejä toteutuneista haavoittuvuuksista ja miksi ne kategorisoidaan loogisiksi haavoittuvuuksiksi.

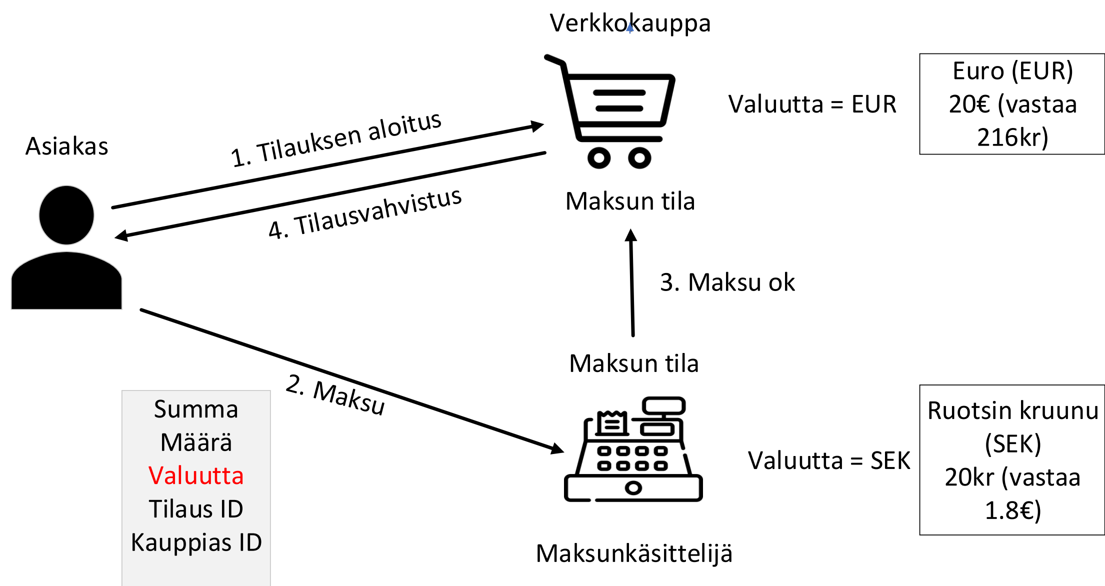
3.1 Yleinen määritelmä

Looginen haavoittuvuus on usein vaikeaa määritellä, koska haavoittuvuustyyppi voi ilmetä monella eri tapaa riippuen sovelluksen tarkoituksesta ja sen myötä toiminnasta. Siksi yksiselitteistä mallia ei ole olemassa ja aiempi tutkimus on pitkälti keskittynyt eri hyökkäyskeinojen kategoriointiin. Voidaksemme puhua aihealueesta laajemmin, tutkielmassa käytetään Deepa ja Thilagam (2016) määritelmää. Heidän mukaansa kaikkia loogisia haavoittuvuuksia yhdistää keino jolla hyökkääjä pystyy manipuloimaan sovelluksen loogista kulkua, johtuen suunnittelu- tai ajatteluvirheestä sovellusta rakentaessa. Ohjelman komponentit toimivat odottamattomalla tapaa ja syntyy yllättävää toiminnallisuutta. Toisin sanoen hyökkäys on mahdollista koska sovelluksen toiminnan säännöt ovat jollakin tapaa viallisia. Tätä ongelmaa voidaan sitten hyödyntää esimerkiksi tietoturva-aukkona sovelluksessa. Loogisten virheiden tapauksissa haavoittuvuus syntyy ja ilmenee osana normaalia sovelluksen toimintaa ja sen myötä hyökkääjä pystyy manipuloimaan validia toiminnallisuutta ilkivaltaisen tavoitteen saavuttamiseksi, vaikka sovellus ilmenisi toimivan täysin oikein.

Tätä voidaan ajatella Nabi, Yong ja Tao (2019) esittämän mallin kautta. Logiikka on kaikkea mitä sovellus tekee. Salasanan palauttaminen, tilin luonti ja verkkokauppaostos ovat kaikki esimerkkejä tästä. Näissä tapauksissa WWW-sovellus odottaa käyttäjän tekevän tietyt askeleet tietyssä vaiheessa. Jos hyökkääjä kumminkin pääsee väärinkäyttämään tai kiertämään näitä komponentteja, pystyy hän mahdollisesti tekemään vahinkoa sovellukselle tai sen käyttäjille. Peruskäyttäjän näkökulmasta sovellus toimii odotetusti, kuten kehittäjä on ajatellutkin, mutta haavoittuvuutta etsivä osapuoli voi löytää ongelman. Yksi haavoitustyypin tunnistamista vaikeuttava ongelma on, että looginen haavoittuvuus on usein näkymätön ja hyök-

kääjä löytää käyttäytymisen erityispiirteen toimimalla sovelluksen kanssa odottamattomalla tavalla joka on näkymätön peruskäyttäjälle. Esimerkiksi muokkaamalla HTTP-kyselyjä tai kiertämällä osa niistä täysin voidaan löytää virheellistä toiminnallisuutta (Pellegrino ja Balzarotti 2014).

Kuviossa 1. tarkastellaan miten looginen haavoittuvuus voisi ilmetä. Käytetään esimerkkinä verkkokauppaa, sen maksunkäsittelijää ja asiakasta, joka tilaa 20 euron hintaisen tuotteen. Asiakas aloittaa tilauksen ja hänet uudelleenohjataan maksunkäsittelijän palveluun, jossa tilaus maksetaan. Tilauksesta siirtyvät tiedot ovat sen summa, tuotteiden määrä, valuutta, sekä tilauksen ja kauppiaan yksilöivät tunnukset. Maksun käsittelyn jälkeen palveluntarjoaja ilmoittaa verkkokaupalle sen olevan kunnossa ja verkkokauppa vahvistaa tilauksen tekemisen asiakkaalle. Esimerkissä on kumminkin ongelma valuutan välittämisessä verkkokaupan ja maksunkäsittelijän välillä. Suunnitteluvirheen takia verkkokauppa käsittelee tuotteita euroissa, mutta tilauksen tietojen siirtyessä maksunkäsittelijälle valuutta käsitellään ruotsin kruunuina. Virheen myötä asiakkaan maksama 20 euroa käsiteltyinä kruunuina vastaa noin 1.8 euroa. Jos verkkokauppa käsittelisi tilauksen automaattisesti ja tuote lähetettäisiin asiakkaalle haavoittuvuus johtaisi taloudelliseen menetykseen.



Kuvio 1. Looginen haavoittuvuus verkkokaupan ja maksunkäsittelijän välillä

3.2 Tarkempi määritelmä

Yleinen määrittely on tärkeä kokonaiskuvan ymmärtämiseksi, mutta spesifien tapauksien kohdalla se tekee käsittelystä vaikeaa. Loogisten haavoittuvuuksien tarkempi kategorisointi on tarpeellista, jotta tiettyjä virheitä on helpompi tunnistaa ja tietää miten korjata ongelma. Tutkielmassa käytetään Deepa ja Thilagam (2016) neljään eri loogisen haavoittuvuuden alakategoriaan jakamista hyökkäysmallista ja kohteesta riippuen, koska tämä tarjoaa helposti käsiteltävän kuvan. Näitä ovat parametrien manipulointi, kulunvalvonnan haavoittuvuudet, sovelluksen kulun ohittaminen ja istunnon hallinnan haavoittuvuudet.

Jos asiakaspuolen validointi ohitetaan ja parametrejä muokataan sovelluksen loogisen virheen takia, puhutaan parametrien manipuloinnista. Syöteparametrien muuttaminen odottamattomaan muotoon ja näiden puuttuva tai viallinen validointi aiheuttaa turvallisuusriskin. Kun palvelin vastaanottaa muutetut parametrit ja käsittelee ne oikeina, voidaan väärinkäyttää sovellusta. Yksinkertaisissa tapauksissa haavoittuvuus voi tapahtua jopa käyttöliittymän kautta. Toinen yleinen tapa on manipuloida HTTP-pyyntöä ja evästeitä (Deepa ja Thilagam 2016). Jos haavoittuvuus sen sijaan kohdistuu käyttäjien valtuuksiin nähdä tiettyjä resursseja, puhutaan kulunvalvonnan haavoittuvuudesta. Kiertämällä todennus tai valtuutus, hyökkääjä pääsee näkemään resursseja, joihin hänellä ei pitäisi olla oikeuksia Esimerkiksi yksityiseksi valittujen tiedostojen tai ylläpitäjän oikeuksiin käsiksi pääseminen on valtuutuksen kiertämistä, jos virhe syntyy loogisen haavoittuvuuden seurauksena. Lähellä tätä on sovelluksen kulun ohittamisen haavoittuvuudet jossa hyökkääjä pääsee kiertämään sovelluksen odotetun työkulun. Odottamattoman resurssin näkemisen sijaan ilkeämielinen käyttäjä pääsee ohittamaan kriittisen polun loogisessa etenemisessä. Nabi, Yong ja Tao (2021) käyttävät tästä esimerkkinä muun muassa tilausvahvistuksen saamista ilman maksamista tai kirjautumissivun kokonaan ohittamista. Kuten voi ajatella, tilauksen lähettäminen ilman maksamista voi aiheuttaa merkittäviä taloudellisia menetyksiä myyjälle. Deepa ja Thilagam (2016) määritelmän viimeisellä istunnon hallinnan haavoittuvuudella tarkoitetaan istunnon parametrien virheellistä käsittelyä. Käytännössä hyökkääjä varastaa toisen käyttäjän istunnon tiedot ja pystyy hyödyntämään tämän oikeuksia. Istunnon varastaminen esimerkiksi ylläpitäjältä tai muulta tärkeältä roolilta voi aiheuttaa merkittävää vahinkoa.

3.3 Reaalimaailman esimerkkejä

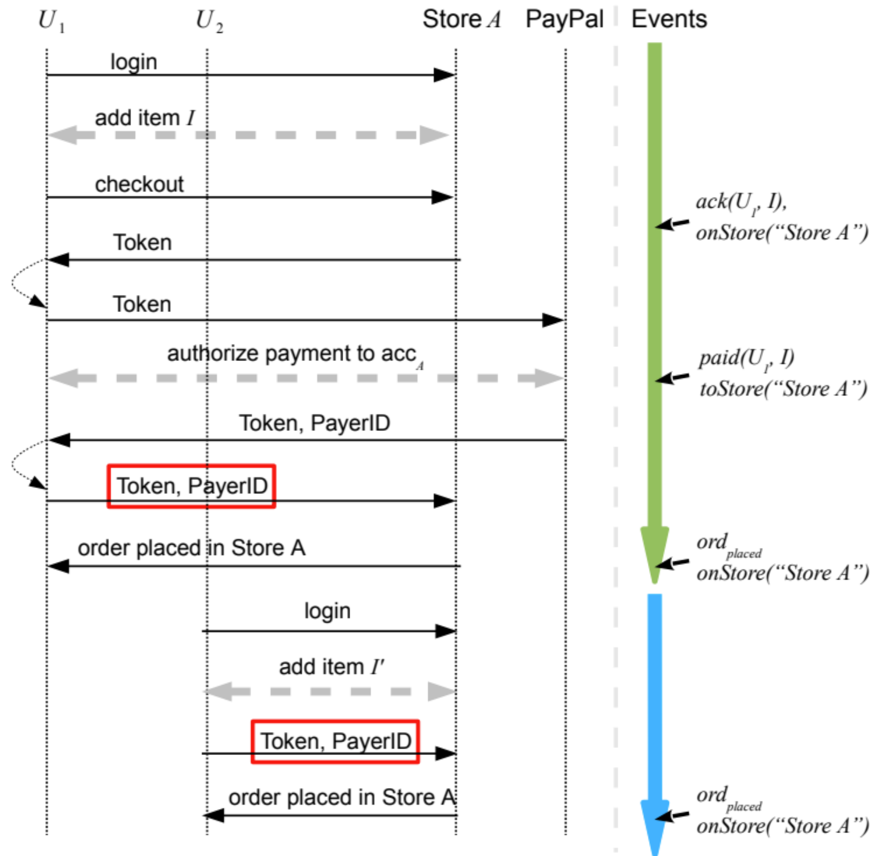
WWW-sovelluksen looginen virhe voi johtaa käyttäjien tietojen vuotamisesta merkittävään taloudelliseen menetykseen. Vaikka haavoittuvuustyyppi vaatii usein kattavaa ymmärrystä ohjelman toiminnasta, on kyseessä oikea reaalimaailman uhka. Vahinkoa on tapahtunut ja tulee tapahtumaan elleivät kehittäjät pyri parantumaan ja seuraamaan parhaita käytänteitä. Tarkempi ymmärrys siitä mitä halutaan tehdä ja miten edetään ohjelman kulussa on tärkeä osa kehitystaitoja. Jokainen ohjelma on uniikki ja sen toiminta vaihtelee sovelluksen mukaan, siksi on hyvä huomioida että ominaisuus joka johtaa tietoturva-aukkoon sovelluksessa ei välttämättä ole sitä toisessa. Looginen haavoittuvuus voi ilmetä monella tapaa ja tarkka kategorisointi on monimutkainen prosessi. Seuraavissa kappaleissa käsitellään kahta eri toteutunutta haavoittuvuutta jotka johtavat erilaisiin lopputilanteisiin. Tästä huolimatta molemmat voidaan kumminkin kategorisoida Deepa ja Thilagam (2016) määrittelemiin alalajeihin.

Kushnir ym. (2021) Käyttää tutkimuksessaan esimerkkinä WordPress sivustoa, jossa käytetään versiota Job Manager lisäosasta jossa on haavoittuvuus. Lisäosa antoi käyttäjien nähdä ja lähettää työtarjouksia. Lisäksi se sisälsi kulunvalvonnan haavoittuvuuden joka toteutui koska pääsynvalvontaa liitetiedostoihin ei oltu toteutettu. Suunnitellessa lisäosaa liitetiedostojen ei oletettu sisältävän mitään arkaluontoista tietoa, tai sitä ei osattu ajatella. Kuka tahansa valtuuttamaton käyttäjä pääsi käsiksi tiedostoihin jotka lähetettiin osana työhakemusta. Kun kyseessä on työnhaku, liitetiedostoina on mahdollista olla muun muassa yksityistä materiaalia. Henkilötiedot kuten nimi, sähköposti, puhelinnumero, yms. annetaan automaattisesti työnantajalle, koska ne ovat tarpeellisia. Kukaan tuskin kumminkaan haluaa mahdollisesti koko WordPress käyttäjäkannan tietävän hänen yksityisiä tietoja. Kun haavoittuvuus havaittiin, sen vakavuudesta johtuen lisäosan käyttö estettiin lokakuussa 2021. Koko elinkaaren aikana se ladattiin kumminkin lähes 200000 kertaa [1].

Pellegrino ja Balzarotti (2014) pyrkivät tutkimuksessaan kehittämään työkalua, jolla voidaan musta laatikko mallisesti havaita loogisia haavoittuvuuksia WWW-sovelluksissa. Yksi työkalulla löydetty haavoittuvuus oli taloudelliseen menetykseen johtava haavoittuvuus. Parametrejä manipuloimalla OpenCart versio 1.5.3.1 ja TomatoCart versio 1.1.7 verkkokauppapalveluiden toimintalogiikasta löytyi virhe, joka on esitetty kuviossa 2. Käyttäjä pystyi ra-

1. katsottu 7.11.2022

kentamaan maksuliikenteessä käytettävän URL:n haluamakseen kahden asiakastilin avulla. Hyödyntämällä muokattua maksuliikennettä hyökkääjä pystyi maksamaan vähemmän kuin tuotteiden hinnaksi oli määritelty. Käyttämällä asiakastilin tunnistetta joka oli maksanut tilauksen, sovelluslogiikka ei tarkistanut onko toinen tili maksanut, jos sen tilaukseen oli yhdistetty alkuperäisen oston tunniste. Sovelluslogiikan mukaan asiakas oli jo maksanut, eikä maksua tarvinut suorittaa uudestaan. Kahdesta tilauksesta myyjä näki kummakin asiakastilin ostokset maksettuina ja valmiina lähetettäväksi. Tästä huolimatta maksu oli suoritettu vain yhdestä tilauksesta. Huolimaton myyjä saattaisi sitten käsitellä tilaukset ja lähettää ne eteenpäin. Haavoittuvuus on korjattu myöhemmissä OpenCart ja TomatoCart palveluiden versioissa.



Kuvio 2. Looginen virhe OpenCart 1.5.3.1 ja TomatoCart 1.1.7 versioissa (Pellegrino ja Balzarotti 2014)

4 Ehkäiseminen

Haavoittuvuuksien tunnistamisessa aiemmat tapahtuneet virheet ja tietoturvarikkeet ovat tärkeä resurssi oppimiseen. Ne auttavat ymmärtämään mistä virhe johtui ja samalla mitä tulevaisuudessa samanlaista ongelmaa vastaan voidaan tehdä. Lisäksi niistä on suuri apu automaattisten työkalujen kehittämisessä. Vaikka työskentely täysin virheettömästi ei tule koskaan olemaan mahdollista, pienikin apu helpottaa työtaakkaa. Kushnir ym. (2021) esittää sovelluskehityksen yhdeksi oleelliseksi komponentiksi automaation. Hänen mukaan ideaalisessa maailmassa tietoturvatestauksen pitäisi olla automatisoitua, jotta kehittäminen pystyisi olemaan tehokasta, jatkuvaa ja toistettavaa. Erityisesti haavoittuvuuksien tapauksissa yksi tapa toteuttaa tämä ovat WWW-sovellusten haavoittuvuusskannerit. Loogisten haavoittuvuuksien tunnistamisessa on kumminkin tiettyjä ongelmia, jotka ovat ominaisia sille.

4.1 Automaattisen havaitsemisen ongelma

Kushnir ym. (2021) määrittelee skannerin työkaluna, jolla voidaan analysoida liikennettä sovelluksen ja käyttäjän välillä. Käytännössä ne toimivat lähettämällä sovelluksen ulkopuolelta erityisesti laadittuja pyyntöjä kohteelle ja analysoimalla vastausta. Normaali käyttötapa on tarkastella mitä käyttäjä lähettää palvelimelle ja millainen vastaus saadaan takaisin, sekä miten vastaus muuttuu parametrejä muunnellessa, esimerkiksi miten sivusto vastaa kirjautumispyyntöön riippuen käyttäjän oikeuksista. Tällaiset skannerit voivat olla kaupallisia tai avoimen lähdekoodin projekteja.

Ongelmana loogisten haavoittuvuuksien tilanteessa on kumminkin vaikeus tunnistaa sovelluksen toteutuksen loogisia komponentteja. Kushnir ym. (2021) käyttää tästä esimerkkinä sitä kuinka skanneri ei tyypillisesti tiedä mitä oikeuksia käyttäjällä on. Esimerkiksi jos peruskäyttäjänä päästään käsiksi ylläpitäjän toiminnallisuuteen, mutta skanneri ei huomaa tätä, ei skanneria voida käyttää luotettavasti. Haavoittuvuus ilmenee oikein toimivan komponentin väärin käyttämisenä, joten skannereilla on vaikeuksia erottaa mistä ongelma syntyy. Sudhodanan ym. (2016) esittää syyksi sen, että useimmat työkalut käyttävät malleja jotka perustuvat tunnettuihin hyökkäystyyppeihin haavoittuvuuden tunnistamiseksi ja siksi esimerkik-

si injektio-tyyppinen haavoittuvuus voidaan tunnistaa tehokkaasti, koska sen komponentit pysyvät osittain samana hyökkäyskohteesta riippumatta. Loogisten haavoittuvuuksien ollessa sovellusspesifisiä ei tiettyä kaavaa ole. Tämän takia automaattiset haavoittuvuusskannerit ovat usein tehottomia loogisten virheiden tunnistamisessa ja vaikka osa automaatiotyökaluista tarjoaa mahdollisuuden skannausominaisuuksien lisäksi, suurin osa ei edes pyri tunnistamaan logiikasta syntyviä ongelmia (Doupé, Cova ja Vigna 2010). Yleistä on sen sijaan käyttää manuaalista lähestymistapaa, jossa testaaja käyttää sieppaavaa välityspalvelinta ja navigoi sovellusta käsin. Näin voidaan tarkemmin analysoida sovelluksen toimintaa ja tunnistaa mahdollisia loogisia heikkouksia ottaen huomioon kehittäjän odotus siitä mitä pitäisi tapahtua tietyissä vaiheissa. Jos virhe löytyy, on haavoittuvuuden väärinkäyttämiseksi on helpompaa ja usein tarpeellista rakentaa myös hyökkäys itse (Sudhodanan ym. 2016). Manuaalisella työllä voidaan varmistaa ettei mitään jää huomioimatta ja ymmärtää tehokaammin mitä sovelluksen on tarkoitus tehdä (Austin ja Williams 2011). Luonnollisesti jos kohdetta ei tunnisteta automaattisesti, ei myöskään hyökkäystä voida toteuttaa niin.

4.2 Menettelytapoja automaatioon

Automaattisen loogisten haavoittuvuuksien tunnistamisen mahdollistamiseksi on kuitenkin yritetty kehittää työkaluja. Kysymys ja ydinongelma lähestymistavoissa on miten saadaan ohjelma ymmärtämään ihmisen ajattelu. Mitä suunnittelija ja kehittäjä ovat ajatelleet luodessaan sovelluksen ja mitä sen kuuluu tavoittaa. Sudhodanan ym. (2016) esittelee yhdeksi ratkaisuvaihtoehdoksi niin sanottua sumeaa logiikkaa käyttävän ratkaisumallin. Työkalut jotka tukevat tällaista eivät etsi vain tiettyä konkreettista selvästi esillä olevaa haavoittuvuutta hyökkäysmalleista. Sen sijaan pyritään löytämään pisteitä, joista on mahdollista löytää haavoittuvuus tai ne vaikuttavat epäilyttävältä toiminnallisuuden kannalta. Pisteitä seuraamalla voidaan päätellä onko todennäköistä, että sovellus sisältää tietoturvaongelman. Tietyn todennäköisyyden ylittyessä merkataan komponentti mahdollisesti haavoittuvaiseksi. Menetelmien haittapuolina on kumminkin suurempi todennäköisyys väärään positiiviseen tai negatiiviseen tulokseen, eli työkalu voi virheellisesti määrittää ominaisuuden haavoittuvaiseksi tai ohittaa oikean ongelman. Täysi automaatio ei ole mahdollista, koska ihmisen työtä vaaditaan määrittämään onko kyseessä oikea virhe.

Felmetsger ym. (2010) Ehdottaa automaattisten loogisten haavoittuvuuksien tunnistamisen mahdollistamiseksi työkalulle tapaa ymmärtää mitä sovellus tekee. Hän käyttää menetelmässään koodin vihjeitä arvioimaan sitä, miten kehittäjä odottaa sovelluksen toimivan. Automaattisella koodin analyysillä katsotaan rajoituksia toiminnalle ja millainen korrelaatio on käyttäjän session datalla ja palvelimen datalla. Tätä verrataan saman työkalun johtamaan sovelluslogiikkaan ilman ihmisen palautetta. Saaduista tuloksista tarkennetaan analyysillä, onko sovelluksessa mahdollisesti haavoittuvuutta. Keinon oletuksena on että tyypillinen käyttäjä hyödyntää sovellusta kehittäjän odotusten mukaan. Monitoroimalla oikeellista liikennettä voidaan päätellä suhteita muuttujien ja rajoitusten tiloihin, sekä komponenttien käyttöjärjestykseen. Ongelmana on kumminkin oletus tietynlaisesta peruskäyttäjästä, joka käyttää sovellusta täysin odotusten mukaan. Jos koodin analyysivaihe ei ole täydellinen, voivat tietyt polut ja komponentit jäädä tarkastamatta.

Yksi lähestymistapa on Kushnir ym. (2021) esittämä käyttäjien oikeuksiin perustuva menettely. Menetelmä olettaa, että käyttäjien roolit voidaan erotella heille esillä olevien elementtien mukaan. Verrattuna yksinkertaiseen peruskäyttäjään, ylläpitäjällä odotetaan olevan laajempi näkymä ja enemmän oikeuksia. Käytännössä mitä painikkeita ja linkkejä voidaan painaa sekä mitä ne tekevät. Käyttäen tätä tietoa ja hyödyntäen hakurobottia, sekä oletuksia erilaisista tavoista olla vuorovaikutuksessa sovelluksen kanssa, voidaan verkkokaapia sivustoa ja arvioida tulisiko ominaisuutta pystyä käyttää. Jos peruskäyttäjänä pystytään pääsemään käsiksi ylläpitäjän ominaisuuksiin on kyseessä haavoittuvuus. Menetelmän heikkoutena on että se tarkastelee vain kulunvalvontaa, eikä huomio muita mahdollisia loogisia haavoittuvuuksia.

Mikään työkalu ei ole täydellinen ja menetelmien ongelmana on ihmisen ajattelusta johtuva rajoite. Koneen on vaikeaa huomata mistä ongelma johtuu, koska se ei ymmärrä tavoitetta. Vaikka olemassa on monia keinoja tunnistaa haavoittuvuustyyppi, yhtä parhaaksi todettua tapaa ei ole löydetty. Tämän takia loogisten haavoittuvuuksien suunnittelijoiden ja kehittäjien harteille jää suuri taakka toimivien ja turvallisten ohjelmien rakentamisessa.

4.3 Kehittäjän vastuu

Kuten Kushnir ym. (2021) määrittelee, että WWW-sovelluksia pitää pyrkiä kehittämään ja niitä täytyy pystyä operoimaan mahdollisimman turvallisesti. Tästä huolimatta haavoittuvuudet tulevat kumminkin todennäköisesti aina olemaan ongelma. Kehittäjä voi luulla olevansa askeleen edellä, mutta mahdollisuus haavoittuvuuden löytymiseen on aina. Siksi on erityisen tärkeää tietää mitä epätoivottuja seurauksia voi aiheutua virheestä kehityksen aikana. Tähän vastuu pitäisi olla sovelluskehittäjällä, sekä erityisesti suunnittelijalla loogisten haavoittuvuuksien tapauksessa.

Kehitysprosessin pitäisi olla koko sovelluksen elinkaaren mittainen. Myöhemmässä vaiheessa havaittujen tietoturva-aukkojen korjaaminen on kalliimpaa kuin varhaisessa vaiheessa (Boehm 1981). Hyvällä työnlaadulla voidaan taata, ettei virheitä ilmene ja vahinkoa ei tapahdu, sekä että ongelmatilanteen toteutuessa kehittäjällä on tarvittavat taidot siitä elpymiseen. Suunnittelua pitäisi pyrkiä toteuttamaan niin, että mahdollisuus haavoittuvuuden syntymiseen pystytään minimoimaan. Kehitystiimin on oltava sen takia tietoinen tietoturva-aukoista, jotka ovat toteutuneet aiemmin. Ammattitaito, koulutus ja ajan tasalla oleminen ovat ratkaisevia tekijöitä turvallisten sovellusten kehittämisessä.

Lisäksi on tärkeää myös muistaa, että vaikka automaattiset työkalut ovat hyviä sovelluksen testausvälineitä, niiden ei voida odottaa tekevän täydellistä työtä. Haavoittuvuus voi syntyä odottamattomalla tapaa jota työkalu ei osaa tunnistaa. Harvat työkalut jotka pystyvät huomioimaan loogisia haavoittuvuuksia ovat usein hyvin tilannekohtaisia. Sovelluksen kehittäjän tulee itsekin ymmärtää mitä tekee, eikä sokeasti luottaa työkaluun. WWW-sovellusten skaalautuessa yhä suuremmaksi kehittäjän itse tulee ymmärtää mitä tekniikoita ja työkaluja hän haluaa ja osaa käyttää. Kuten McGraw (2006) sanoo kirjassaan: "Turvallisuusongelmat kehittyvät, kasvavat ja muuntuvat aivan kuten lajit mantereella. Mikään tekniikka tai sääntökokonaisuus ei koskaan pysty täydellisesti havaitsemaan kaikkia tietoturva-aukkoja."

Vaikka loogisten virheiden tunnistaminen on kehittynyt viime vuosikymmenen aikana, on silti paljon tehtävää ja alue on kokonaisuudessaan vähän tutkittu. Kuten Deepa ja Thilagam (2016) mainitsevat, iso osa valmiista tutkimuksesta tarkastelee vain yhtä haavoittuvuustyypin alakategoriaa tai miten haavoittuvuustyyppi voidaan havaita automaattisesti. Sovelluk-

sien loogisten haavoittuvuuksien kokonaisuuden tunnistaminen on siksi vaikeaa. Monella tapaa ilmeneminen ja sovelluskohtaisuus tekee yhdeksi kokonaiseksi käsitteeksi kasaamisesta hankalaa. Kehittäjän on helpompaa ymmärtää mistä ongelma johtuu, jos se on helppo määritellä. Epäselvissä tapauksissa, kuten loogisissa haavoittuvuuksissa tarvitaan taitavaa henkilöä ja ymmärrystä mitä tavoitellaan sovelluksella. Sama koskee virheen tapahtuessa. On tärkeää ymmärtää mikä komponentti aiheutti tietoturvaongelman ja mistä se johtuu. Kyse on harvoin huonosta koodista, ongelma sen sijaan syntyy osana suunnittelua tai sen toteutusta. Komponenttien yhteisen toiminnan ymmärtämiseen tarvitaan siksi ammattitaitoisen ihmisen apua.

5 Yhteenveto

Haavoittuvuudet ovat jatkuva ongelma sovelluskehityksessä. Virhe sovelluskoodissa voi johtaa moneen eri tilanteeseen ja haavoittuvuuteen. Looginen haavoittuvuus on esimerkki tästä. Hyökkääjä voi aiheuttaa merkittävää vahinkoa arkaluontoisten tietojen vuotamisesta taloudelliseen menetykseen tai ketjuttaa haavoittuvuuden vielä vakavampaan ongelmaan. Siksi on tärkeää ymmärtää miten ne ilmenevät ja mistä ongelma syntyy. Kehittäjän tulisi pystyä tunnistamaan ongelmia jo suunnitteluvaiheessa, koska loogisen haavoittuvuuden korjaaminen vaatii usein merkittäviä muutoksia ohjelmakoodiin.

Koska looginen haavoittuvuus voi ilmetä monella eri tapaa ja kattoterminä se voidaan jakaa moneen alakategoriaan, on määrittely ja löytäminen usein monimutkaista, erityisesti automaattisilla työkaluilla. Ongelma haavoittuvuusskannerien käytössä on niiden toimintamenetelmissä. Ne tunnistavat loogisen haavoittuvuuden usein ominaisuutena sovelluksessa, joka toimii odotetulla tapaa. Ongelman korjaamiseksi työkalun tulisi ymmärtää miten sovelluksen odotetaan toimivan. Tähän on esitettyjä keinoja, mutta ne ovat usein tapauskohtaisia yhden tyyppiseen loogiseen haavoittuvuuteen eivätkä siksi tehokkaita.

Loogiset haavoittuvuudet WWW-sovelluksissa ovat monimutkainen ongelma, jonka korjaamiseksi ei ole yhtä oikeaa ratkaisua tai edes lähestymistapaa. Haavoittuvuustyyppi on vaikea löytää ja korjata, koska ongelma on pohjimmillaan sovelluskehityksen suunnitteluvaiheessa. Ihminen on siksi paras työkalu loogisen haavoittuvuuden havaitsemiseen ja korjaamiseen. Turvallisen sovelluksen kehittämiseksi tarvitaan tuntemusta ohjelman toiminnasta ja loogisesta etenemisestä, jos halutaan välttää haavoittuvuuden syntyminen.

Lähteet

Amoroso, E.G. 1994. *Fundamentals of Computer Security Technology.* PTR Prentice Hall. ISBN: 9780131089297. https://books.google.fi/books?id=f95QAAAAMAAJ.

Anley, Chris. 2002. “Advanced SQL injection in SQL server applications”, https://priv.gg/e/Hacking%20-%20Advanced%20SQL%20Injection.pdf.

Apostu, Anca, Florina Puican, Geanina Ularu, George Suciu, Gyorgy Todoran ym. 2013. “Study on advantages and disadvantages of Cloud Computing–the advantages of Telemetry Applications in the Cloud”. *Recent advances in applied computer science and digital services* 2103. https://d1wqtxts1xzle7.cloudfront.net/53325989/cloudcomputing-pros_cons-with-cover-page-v2.pdf?Expires=1669627431&Signature=ZnJESkw~tIxtVzLRFkoTWj3YYhKMte7LBrZRNuMofk7SkgR6h1X5plQooR4~jrC57gzPyugpUQsSR0Z3ua6KAxmY5Y5f9cbQnbhVvwTsxU8toqbS4CUWXl4rSOXE9RgP8sSJcDgJV00pAIv-CL8pC9AlJRE-0eFDfi3cFc9Ihzyhy2Rqr86kjFr4qHDLU4HdgZUr4D4GJ0TY5vX9vNyvGVCboZMSLfgdVVd9eVxZb3JWCAyoS-sJbqtpdg2kShUSHjb3CEk1~cWDijrQPprXdBNRGr~jUYQZt8nEZUH9DXAtRywLgglEUche2iehB-e3czEG5LUEcVlTT6RDWEwINg__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.

Aslam, Taimur, Ivan Krsul ja Eugene H Spafford. 1996. “Use of a taxonomy of security faults”, https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=2304&context=cstech.

Austin, Andrew, ja Laurie Williams. 2011. “One technique is not enough: A comparison of vulnerability discovery techniques”. Teoksessa *2011 International Symposium on Empirical Software Engineering and Measurement,* 97–106. IEEE. https://www.sjoerdlangkemper.nl/papers/2011/one-technique-is-not-enough-a-comparison-of-vulnerability-discovery-techniques-austin-williams.pdf.

Bau, Jason, Elie Bursztein, Divij Gupta ja John Mitchell. 2010. “State of the art: Automated black-box web application vulnerability testing”. Teoksessa *2010 IEEE symposium on security and privacy,* 332–345. IEEE. http://crypto.stanford.edu/~jcm/papers/pci_oakland10.pdf.

Boehm, B.W. 1981. *Software Engineering Economics.* Prentice-Hall advances in computing science and technology series. Prentice-Hall. ISBN: 9780138221225. https://books.google.fi/books?id=mpZQAAAAMAAJ.

Boyer, Robert S, ja J Strother Moore. 1979. *A computational logic.* Academic Press. https://books.google.fi/books?hl=fi&lr=&id=OjijBQAAQBAJ&oi=fnd&pg=PP1&dq=Boyer,+R.S.+and+Moore,+J+S.,+A+Computational+Logic,+Academic+Press,+1979.&ots=W07B6Lkggz&sig=YUGOU1299DuvIOAGCoTExuMwdJs&redir_esc=y#v=onepage&q&f=false.

Conallen, Jim. 1999. "Modeling web application architectures with UML". *Communications of the ACM* 42 (10): 63–70. https://dl.acm.org/doi/pdf/10.1145/317665.317677.

Deepa, G, ja P Santhi Thilagam. 2016. "Securing web applications from injection and logic vulnerabilities: Approaches and challenges". *Information and Software Technology* 74:160–180. http://muhammetbaykara.com/wp-content/uploads/2018/10/ymhgece_webvul.pdf.

Doupé, Adam, Marco Cova ja Giovanni Vigna. 2010. "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners". Teoksessa *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment,* 111–131. Springer. https://www.researchgate.net/profile/Adam-Doupe/publication/221394405_Why_Johnny_Can't_Pentest_An_Analysis_of_Black-Box_Web_Vulnerability_Scanners/links/02bfe510830b54f61d000000/Why-Johnny-Cant-Pentest-An-Analysis-of-Black-Box-Web-Vulnerability-Scanners.pdf.

Eichmann, David. 1994. "The RBSE spider-balancing effective search against web load". Teoksessa *Proc. 1st WWW Conf.* Citeseer. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0c43c9c53ea668589b4dfc260ddb7fa601aa3b78.

Felmetsger, Viktoria, Ludovico Cavedon, Christopher Kruegel ja Giovanni Vigna. 2010. "Toward automated detection of logic vulnerabilities in web applications". Teoksessa *19th USENIX Security Symposium (USENIX Security 10).* https://www.usenix.org/legacy/events/sec10/tech/full_papers/Felmetsger.pdf.

Gellersen, H-W, ja Martin Gaedke. 1999. "Object-oriented web application development". *IEEE Internet Computing* 3 (1): 60–68. https://www.researchgate.net/profile/Martin-Gaedke/publication/3419231_Object-Oriented_Web_Application_Development/links/570d215b08ae3199889bb883/Object-Oriented-Web-Application-Development.pdf.

Kushnir, Malte, Olivier Favre, Marc Rennhard, Damiano Esposito ja Valentin Zahnd. 2021. "Automated black box detection of HTTP GET request-based access control vulnerabilities in web applications". Teoksessa *ICISSP 2021, online, 11-13 February 2021,* 204–216. SciTePress. https://www.researchgate.net/publication/349405236_Automated_Black_Box_Detection_of_HTTP_GET_Request-based_Access_Control_Vulnerabilities_in_Web_Applications.

Leveson, Nancy, ym. 1995. "Medical devices: The therac-25". *Appendix of: Safeware: System Safety and Computers,* https://cs.ucf.edu/~dcm/Teaching/COP4600-Spring2011/Literature/Therac25-Leveson.pdf.

Linde, Richard R. 1975. "Operating system penetration". Teoksessa *Proceedings of the May 19-22, 1975, national computer conference and exposition,* 361–368. https://dl.acm.org/doi/abs/10.1145/1499949.1500018.

McGraw, G. 2006. *Software Security: Building Security in.* Addison-Wesley professional computing series. Addison-Wesley. ISBN: 9780321356703. https://www.garymcgraw.com/technology/software-security/.

Nabi, Faisal, Jianming Yong ja Xiaohui Tao. 2019. "A Taxonomy of Logic Attack Vulnerabilities in Component-based e-Commerce System", https://infonomics-society.org/wp-content/uploads/A-Taxonomy-of-Logic-Attack-Vulnerabilities-in-Component-based-e-Commerce-System.pdf.

———. 2021. "Classification of Logical Vulnerability Based on Group Attack Method". *J. Ubiquitous Syst. Pervasive Networks* 14 (01): 19–26. https://iasks.org/articles/juspn-v14-i1-pp-19-26.pdf.

Pellegrino, Giancarlo, ja Davide Balzarotti. 2014. "Toward Black-Box Detection of Logic Flaws in Web Applications." Teoksessa *NDSS,* 14:23–26. https://www.ndss-symposium.org/wp-content/uploads/2017/09/04_2_1.pdf.

Sudhodanan, Avinash, Alessandro Armando, Roberto Carbone, Luca Compagna ym. 2016. “Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications.” Teoksessa *NDSS.* https://www.ndss-symposium.org/wp-content/uploads/2017/09/attack-patterns-black-box-security-testing-multi-party-web-applications.pdf.

Zadeh, Lotfi A. 1988. “Fuzzy logic”. *Computer* 21 (4): 83–93. https://ieeexplore.ieee.org/abstract/document/53.