TURUN YLIOPISTO

XSS-haavoittuvuudet ja niiden havaitseminen web-sovelluksissa

Ohjelmistotekniikka
Tieto- ja viestintätekniikan tutkinto-ohjelma
Tietotekniikan laitos, Teknillinen tiedekunta
TkK-tutkielma

Laatija:
Mikael Niku-Paavola

Ohjaaja:
Sampsa Rauti (Turun yliopisto)

Huhtikuu 2024

TkK-tutkielma
Tietotekniikan laitos, Teknillinen tiedekunta
Turun yliopisto

Oppiaine: Ohjelmistotekniikka
Tutkinto-ohjelma: Tieto- ja viestintätekniikka
Tekijä: Mikael Niku-Paavola
Otsikko: XSS-haavoittuvuudet ja niiden havaitseminen web-sovelluksissa
Sivumäärä: 22 sivua
Päivämäärä: Huhtikuu 2024

Internet ja web-sovellukset ovat olennainen osa nykymaailmaa ja jatkuvan suosion myötä niiden tietoturva on yhä tärkeämmässä asemassa. Cross-site Scripting (XSS) on yksi yleisimmistä web-sovellusten haavoittuvuuksista. XSS-haavoittuvuutta hyödyntävällä hyökkäyksellä voidaan muokata web-sivuston sisältöä mielivaltaisesti, kaapata käyttäjän selainistunto tai varastaa käyttäjätietoja. Tässä tutkielmassa käsitellään menetelmiä XSS-haavoittuvuuksien havaitsemiseen web-sovelluksissa. Jos haavoittuvuudet voidaan havaita ajoissa, ne voidaan korjata ennen kuin niitä hyväksikäytetään hyökkäyksissä.

Tutkielmassa käsitellään ja vertaillaan XSS-haavoittuvuuksien havaitsemismenetelmiä. Tutkimus toteutetaan kirjallisuuskatsauksena. Menetelmät on jaettu niiden käyttämän analyysimenetelmän perusteella kolmeen kategoriaan, jotka ovat staattinen analyysi, dynaaminen analyysi ja hybridianalyysi. Käsitellyillä menetelmillä on mahdollista havaita XSS-haavoittuvuuksia web-sovelluksissa tarkasti ja kattavasti. Uusimmissa menetelmissä hyödynnetään syväoppimista, vahvistusoppimista ja geneettisiä algoritmeja. Menetelmien kyky havaita haavoittuvuuksia on parantunut jatkuvasti vuoden 2015 jälkeen. Menetelmät ovat edistyneitä, mutta ne ovat usein rajoittuneita toimimaan vain tiettyjen ympäristöjen tai ohjelmointikielten kanssa. Lisäksi monet menetelmistä kykenevät havaitsemaan vain yhdenlaisia XSS-haavoittuvuuksia. Näihin ongelmiin olisi hyvä pyrkiä löytämään ratkaisuja tulevaisuudessa.

Asiasanat: tietoturva, web-sovellukset, XSS, haavoittuvuuksien havaitseminen

Sisällysluettelo

1 Johdanto

Cross-site scripting (XSS) on web-sovelluksissa esiintyvä haavoittuvuus. Tämä haavoittuvuus mahdollistaa mielivaltaisen koodin injektoimisen syötteen kautta sivuston sisältöön. Tämä koodi on yleensä JavaScript-kielellä kirjoitetun skriptin muodossa. Tämä taas saattaa mahdollistaa hyökkäyksen sivustoa tai sen käyttäjiä vastaan. XSS on vuodesta toiseen ollut yleisimpien web-sovelluksissa esiintyvien haavoittuvuuksien joukossa. Open Worldwide Application Security Project (OWASP):n julkaisemalla ”OWASP Top Ten” -nimisellä yleisimpien haavoittuvuuksien listalla XSS oli vuonna 2017 sijalla 7 ja vuonna 2021 sijalla 3 (OWASP, 2021). Tosin vuoden 2021 listauksessa XSS oli kaikkien injektiohaavoittuvuuksien kanssa saman nimikkeen alla. On siis selvää, että XSS on hyvin keskeinen käsite liittyen web-sovellusten tietoturvaan (Cui Y. ja muut 2020). Tämän tutkielman tarkoituksena on selvittää menetelmiä, joilla XSS haavoittuvuuksia voidaan etsiä web-sovelluksista. Keskeisin tutkimuskysymys, johon tämä tutkielma pyrkii vastaamaan, on siis seuraava:

Millä menetelmillä XSS-haavoittuvuuksia voidaan havaita web-sovelluksissa?

Kirjallisuuskatsauksessa mukailtiin osittain Kitchenhamin ja muiden (2009) esittämää systemaattista kirjallisuuskatsausta. Tiedonhaku suoritettiin Turun yliopiston Volter-tietokannassa. Hakusana oli *(XSS OR "Cross?site Scripting") AND ("vulnerability detection" OR "detecting vulnerabilit*" OR "detection of vulnerabilit*") AND web**. Haku rajattiin vuonna 2015 tai myöhemmin julkaistuihin englanninkielisiin tuloksiin. Haku tuotti 73 tulosta, joista valittiin otsikon perusteella aiheeseen sopivaksi 29 tulosta. Abstraktin sisällön perusteella näistä valittiin edelleen 25 julkaisua. Koko sisällön sopivuuden ja olennaisuuden perusteella lopulliseen aineistoon valikoitui 6 julkaisua. Myös joitain näissä julkaisuissa käytetyistä lähteistä valittiin aineistoon.

Luvussa 2 käydään läpi Cross-site Scripting -haavoittuvuuksien taustaa ja keskeisiä käsitteitä. Aliluvussa 2.1 käsitellään XSS-haavoittuvuuksien eri kategoriat ja ominaisuudet. Luvussa 2.2 käydään läpi XSS-haavoittuvuuksia hyödyntäviin hyökkäyksiin usein liittyvien evästeiden taustaa. Lopuksi luvussa 2.3 esitetään käytännön esimerkkejä erityyppisiä XSS-haavoittuvuuksia hyödyntävistä hyökkäyksistä.

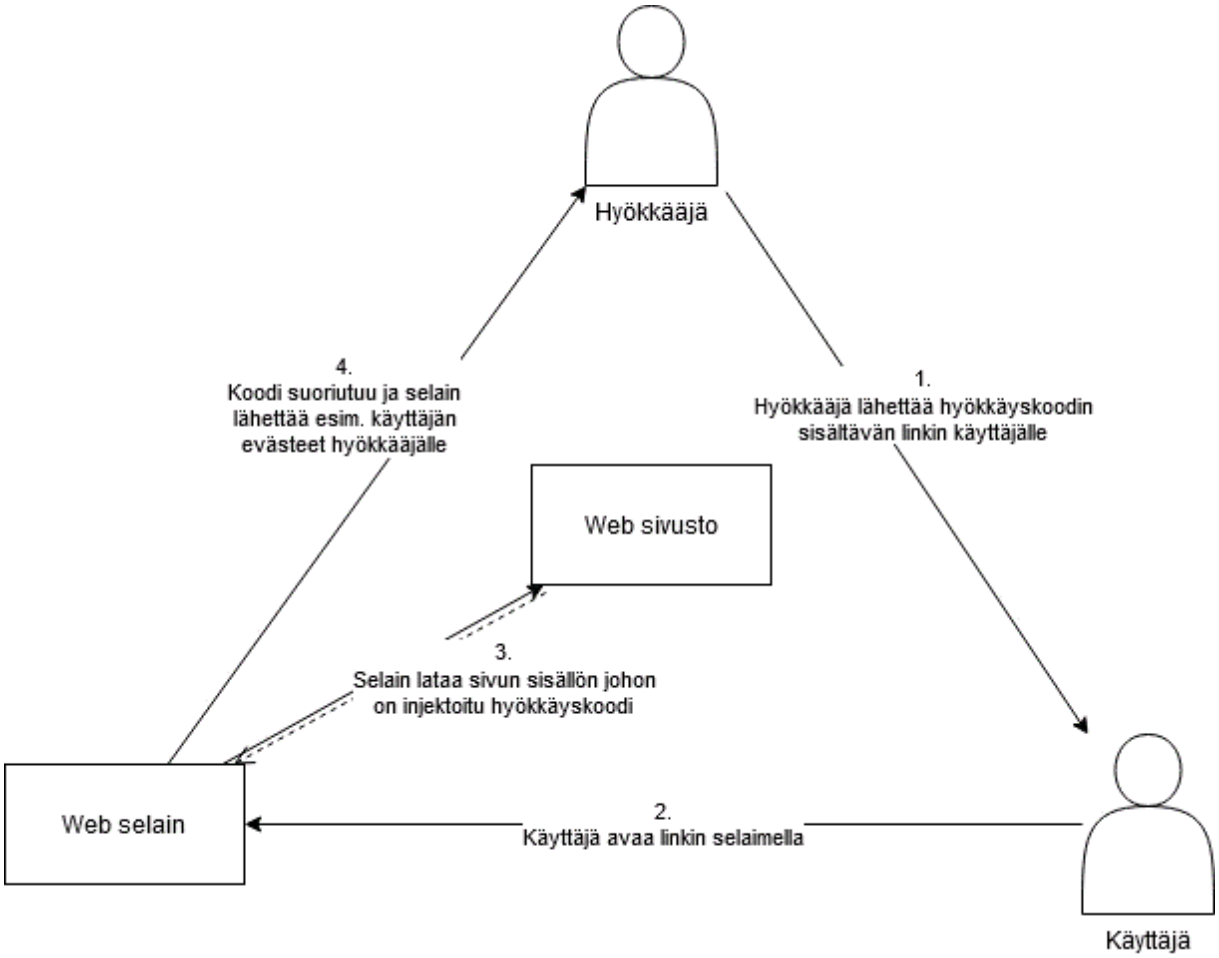
Luvussa 3 käsitellään tutkimusaineiston avulla menetelmiä XSS-haavoittuvuuksien havaitsemiseen. Luvussa 3.1 kerrotaan staattista analyysia käyttävistä menetelmistä. Luvussa 3.2 tarkastellaan dynaamista analyysia hyödyntäviä menetelmiä. Luvussa 3.3 käydään läpi

staattista ja dynaamista analyysia yhdistäviä hybridianalyysimenetelmiä. Luvussa 4 on pohdintaa aineiston pohjalta ja sen analysointia. Luku 5 sisältää yhteenvedon aikaisemmin käsitellyistä asioista.

2 Cross site scripting

Cross-Site Scripting (XSS) on verkkosivujen haavoittuvuus, jota hyödyntämällä hyökkääjä voi injektoida haitallista JavaScript-koodia verkkosivulle. Injektoitu koodi suoritetaan uhrin selaimessa. Tämä voi johtaa erilaisiin haitallisiin toimintoihin, kuten sivuston sisällön muokkaamiseen, istuntojen varastamiseen tai käyttäjän ohjaamiseen haitallisille sivustoille. XSS-haavoittuvuudet olivat olemassa jo World Wide Webin (web) alkuaikoina vuonna 1996. JavaScript (JS) -nimisen ohjelmointikielen saapuminen toi mukanaan myös XSS-haavoittuvuudet ja muutti web-sovellusten maailman pysyvästi. JavaScript mahdollisti interaktiivisien ominaisuuksien kuten pudotusvalikkojen ja ponnahdusikkunoiden sisällön muokkaamisen dynaamisesti ilman, että sivua täytyy ladata kokonaan uudestaan. Se kuitenkin myös antoi hakkereille täysin uudenlaisia mahdollisuuksia hyökkäyksille. Uhrin saapuessa hakkerin luomalle sivulle he pystyivät pakottamaan sivun lataamaan minkä tahansa web-sivuston samaan selaimen ikkunaan. Käyttäen JavaScriptiä he pystyivät sitten yhdistämään sivut toisiinsa ja siirtämään tietoa toiselta sivulta toiselle. Näin oli mahdollista varastaa käyttäjien käyttäjänimiä, salasanoja, evästeitä (engl. cookie) ja mitä tahansa muuta luottamuksellista tietoa johon uhrilla oli pääsy. (Grossman 2007.)

XSS-haavoittuvuutta hyödyntäen on mahdollista suorittaa hyökkäys, joka pakottaa web-sivuston esittämään mielivaltaista koodia, jonka uhrin selain sitten suorittaa. XSS-koodi, joka on tyypillisesti kirjoitettu Hypertext Markup Language (HTML) tai JavaScript-kielellä, ei suoritu palvelimella. Palvelin toimii isäntänä, mutta hyökkäyksen suoritus tapahtuu asiakaspuolella web-selaimessa. Hyökkääjä käyttää luotettua sivustoa alustana hyökkäykselle, mutta tavoiteltu uhri ei ole palvelin itsessään vaan sen käyttäjä. (Grossman 2007.) Tästä huolimatta, jos hyökkääjä saa XSS-hyökkäyksen seurauksena käsiinsä esimerkiksi ylläpitäjän käyttäjätunnukset, voi hän sen avulla kohdistaa hyökkäyksiä myös palvelinta ja sovellusta kohtaan. Alla oleva kuva 1 havainnollistaa hyökkäystä ja sen etenemistä.



Kuva 1: XSS-hyökkäyksen eteneminen. Hyökkäys on tyypiltään heijastettu XSS-hyökkäys.

Kuten kuvassa 1 esitetään, hyökkäys alkaa, kun hyökkääjä on valmistellut hyökkäyskoodinsa ja lähettää hyökkäyskoodin sisältävän linkin käyttäjälle. Pahaa aavistamaton käyttäjä avaa linkin luotetulle sivustolle tietämättään hyökkäyskoodista. Käyttäjän selain yhdistää web-sivustoon ja lataa pyydetyn sivun sisällön, joka sisältää myös hyökkääjän injektoiman koodin. Kun selain on ladannut sivun, injektoitu hyökkäyskoodi suoriutuu selaimessa. Hyökkääjän tavoitteesta riippuen koodin suorituksen seurauksena selain lähettää esimerkiksi uhrin käyttäjätiedot, evästeet tai selainistunnon hyökkääjälle. Kuvan 1 esittämä heijastettu XSS-hyökkäys on yksi hyökkäyksen tyypeistä, joita käsitellään seuraavaksi.

2.1 XSS hyökkäysten tyypit

XSS-hyökkäykset voidaan jakaa kolmeen luokkaan, jotka ovat pysyvä (engl. persistent), ei-pysyvä (engl. non-persistent) ja Document Object Model (DOM) pohjainen (engl. DOM based) XSS. Ei-pysyvästä XSS:stä käytetään myös yleisesti nimeä heijastettu (engl. reflected) XSS ja pysyvästä XSS:stä nimitystä tallennettu (engl. stored) XSS. XSS-hyökkäykset hyödyntävät

puutteita sivustolla olevien syötteiden suodattamisessa ja validoinnissa, mikä mahdollistaa kokonaisten skriptien lähettämisen ja injektoimisen sivulla suoritettavan koodin joukkoon. Nämä skriptit saatetaan tallentaa tekstitiedostoihin komentoina, joka tulkitaan rivi riviltä reaaliajassa suoritusta varten. (Rodríguez ja muut 2020.)

Heijastettu XSS-hyökkäys hyödyntää haavoittuvuuksia web-sovelluksissa, jotka käyttävät (heijastavat) käyttäjän syöttämää tietoa sivun sisältönä. Yleisiä kohteita ovat hakukoneet ja -kentät, lomakkeet, URL-osoitteet, evästeet tai jopa videot. (Rodríguez ja muut 2020.) Hyökkääjä etsii ensin XSS-haavoittuvuuden sivustolta, jonka jälkeen hyökkääjä rakentaa hyökkäykseen tarkoitetun URL-osoitteen. Saatuaan hyökkäysohjelmansa valmiiksi hyökkääjä levittää tätä erityistä URL-linkkiä uhreille esimerkiksi sähköpostin, keskustelufoorumien, viestien ja muiden väylien kautta. Tämä hyökkäys on erityisen tehokas, sillä hyökkääjän URL sisältää oikean web-sivuston osoitteen. Uhrit todennäköisemmin avaavat tällaisen linkin toisin kuin valeosoitteen tai satunnaisen IP-osoitteen sisältävän linkin, joita käytetään usein huijausviesteissä. (Grossman 2007.) Heijastetun XSS-hyökkäyksen tavoitteena on usein kaapata uhrin evästeet, joiden avulla hyökkääjä voi esittäytyä uhrina. Evästeiden avulla hyökkääjä voi suorittaa toimintoja uhrin oikeuksilla ilman minkäänlaista salasanaa eli kaapata uhrin istunnon. (Rodríguez ja muut 2020.)

Tallennetut XSS-hyökkäykset hyödyntävät usein heikkouksia foorumi- ja blogisivustoilla (Cui ja muut 2020). Hyökkääjä sijoittaa haitallisen koodinsa sivustolle lähetettäviin viesteihin ja toimituksiin kuten kommentteihin, arvosteluihin, julkaisuihin ja chat-viesteihin (Grossman 2007). Nämä viestit ja niihin sijoitetut haitalliset ohjelmat tallentuvat web-sovelluksen tietokantaan (Cui ja muut 2020). Tallennettu XSS ei vaadi erityisesti luotua linkkiä koodin suorittamiseen. Hyökkääjä yksinkertaisesti sijoittaa XSS-koodin paikkaan web-sivustolla, jolla käyttäjät todennäköisesti käyvät. Kun käyttäjä avaa haitallisella koodilla saastutetun (engl. infected) sivun sivustolla, koodi suorittuu automaattisesti. Tästä syystä tallennettu XSS on huomattavasti vaarallisempi hyökkäys verrattuna heijastettuun tai DOM-pohjaiseen XSS-hyökkäykseen. Käyttäjällä ei ole mitään tapaa suojata itseään hyökkäykseltä. Kun hyökkääjä on sijoittanut haitallisen koodinsa sivustolle, hän mainostaa saastutettua sivua houkutellakseen mahdollisia uhreja. Jopa käyttäjät, jotka tunnistaisivat hyökkäykseen liittyvän URL-linkin, saattavat helposti päätyä tallennetun XSS-hyökkäyksen uhriksi. (Grossman 2007.)

DOM-pohjainen XSS-hyökkäys kohdistuu palvelinpuolen koodin sijaan haavoittuvaan käyttäjäpuolen (engl. client-side) koodiin. Hyökkääjä luo haitallista JavaScript koodia

sisältävän linkin ja lähettää sen uhrille. Avatessaan linkin uhri saa palvelimelta vastauksen, joka ei sisällä haitallista koodia. Haitallinen koodi suorittuu käyttäjäpuolella ja hyökkääjä voi näin päästä käsiksi käyttäjän henkilökohtaisiin tietoihin. (Cui ja muut 2020.) DOM XSS on muita hyökkäyksiä vaikeampi havaita sillä haitallinen koodi ei saavuta palvelinta. Hyökkäystä myös pidetään monimutkaisempana ja tuntemattomampana kahteen muuhun luokkaan verrattuna. (Rodríguez ja muut 2020.)

2.2 Evästeet

Evästeiden (engl. cookie) kaappaaminen on hyvin yleinen XSS-hyökkäyksen tavoite. Termi on myös varmasti tuttu suurelle osalle nykyajan ihmisistä, mutta käsitteen ymmärtäminen on olennaista XSS-hyökkäysten ymmärtämiseksi. Evästeet ovat käyttäjän ja web-sovelluksen välisen istunnon (engl. session) ylläpitämiseen käytetty mekanismi, joka nykyään löytyy käytännössä lähes oletuksena web sovelluksista. Niiden käytössä on kuitenkin synnynnäisiä haavoittuvuuksia, jotka mahdollistavat luotujen istuntojen eheyttä uhkaavat hyökkäykset. Turvatun protokollan, kuten HTTPS:n käyttäminen on suositeltavaa evästeiden suojaamiseksi. Tämä kuitenkin lisää sovelluksen toteutuksen ja ylläpidon kustannuksia sekä heikentää sovelluksen suorituskykyä. Lisäksi evästeihin on mahdollista päästä käsiksi, vaikka käytettäisiinkin HTTPS-protokollaa, sillä protokolla suojaa vain verkkoyhteyden. (Rodríguez ja muut 2020.)

Evästeet tallennetaan kevytrakenteisiin tietokantoihin (engl. lightweight databases). Esimerkiksi Mozilla Firefox -selain käyttää evästeiden tallentamiseen SQLite-formaattia. Evästeiden tarkoitus on säilyttää olennaista tietoa muun muassa selauskäytännöistä, istunnoista, pääsytiedoista, vierailluista sivustoista ja vierailukertojen tiheydestä. Näihin tietoihin voi sisältyä esimerkiksi millaista web selainta tai laitetta sivuston selaamiseen käytettiin. Toisin sanoen evästeet toimivat ikään kuin web-sivustojen muistina. Evästeiden avulla voidaan asettaa käyttäjän asetukset ja mieltymykset, jotta käyttäjän käyttäjäkokemus olisi parempi seuraavalla vierailukerralla. (Rodríguez ja muut 2020.)

Lisäksi evästeitä käytetään myös mainostuksessa. Web-sivustoilla näkyvät mainokset tulevat mainossivustoilta tai kolmansilta osapuolilta. Käyttäjän tietoja välitetään web-sivuston ja mainoksia tarjoavan tahon välillä. Yksi esimerkki tällaisesta tahosta on Facebook (Meta). Tästä ilmenee selvä ongelma käyttäjän yksityisyyden kannalta. Evästeiden sisältöä tutkiessa käyttäjä ei kykene arvioimaan ovatko evästeet vaarattomia vai lähettävätkö ne henkilökohtaista tietoa

kolmannelle osapuolelle. Käyttäjälle ei jää muuta vaihtoehtoa kuin luottaa käyttämäänsä web-sivustoon ja kolmanteen osapuoleen sekä sivuston maineeseen. (Rodríguez ja muut 2020.)

Evästeiden käytön yleisyys ja niiden tietoturvan luontainen heikkous ovat vaarallinen yhdistelmä. Tämän ongelman ratkaisemiseksi on kuitenkin kehitetty erilaisia menetelmiä, jotka voidaan lähestymistapansa osalta jakaa karkeasti kahteen ryhmään. Ensimmäinen ryhmä keskittyy evästeiden tietoturvan parantamiseen, kun taas toinen evästeiden kaappaamisen estämiseen. Tietoturvan parantamiseksi evästeisiin esimerkiksi lisättiin mahdollisuus asetuksille (engl. attribute) HttpOnly ja Secure. Secure-asetuksen käyttö rajoittaa evästeen ainoastaan turvatuille kanaville. Tämä asetus viestii asiakasohjelmalle, että eväste tulee liittää vain viesteihin, jotka lähetetään käyttäen Secure Socket Layer (SSL) tai Transport Layer Security (TLS) -protokollaa. Asiakasohjelma kuitenkin lähettää viestin, jos hyökkääjä tekee pyynnön turvatulta sivustolta. HttpOnly-asetus taas rajoittaa evästeen vain HTTP pyyntöihin. Toisin sanoen asiakasohjelma liittää evästeen pyyntöön vain, jos kyseessä on HTTP-pyyntö. Tämä asetus kehitettiin erityisesti estämään evästeiden kaappaaminen XSS-hyökkäyksellä. Se estää JavaScript koodia pääsemästä käsiksi evästeeseen ja myös estää evästeen pääsyn rajapintoihin, jotka eivät käytä HTTP-protokollaa. HttpOnly-asetus ei kuitenkaan yksinään täysin poista XSS-hyökkäysten uhkaa, eikä se suojaa kuin osaa evästeistä. (Choi ja muut 2019.)

2.3 Esimerkkejä hyökkäyksistä

Havainnollistavana esimerkkinä **heijastetusta XSS-haavoittuvuudesta** käytetään Open Worldwide Application Security Projectin (OWASP) ylläpitämää tarkoituksellisesti haavoittuvaa Juice Shop web-sovellusta.[1] Sovelluksen hakutoiminto sisältää XSS-haavoittuvuuden, jota voidaan hyödyntää heijastetun XSS-hyökkäyksen suorittamiseen. Tätä haavoittuvuutta hyödyntäen esitetään hyökkäys, jossa uhrin selainistunto kaapataan varastamalla evästeet. Voimme huomata, että hakukenttään syötetty hakutermi heijastuu sivun HTML-sisällössä, sekä osana URL-osoitetta. Sivusto käyttää jonkinlaista suodatusta hakufunktiossa, mutta vain tietyt HTML-tagit kuten `<script></script>` ovat suodatettu. Hyödynnetään puutteellista suodatusta ja syötetään hakuun seuraava HTML-elementin: `<img`

[1] OWASP:n ylläpitämän Juice Shop sivuston osoite: https://juice-shop.herokuapp.com

```
src=1   onerror=window.open("https://webhook.site/92da0c6d-30d2-4d08-9927-de9d58da48f9/?cookie="+document.cookie)>
```

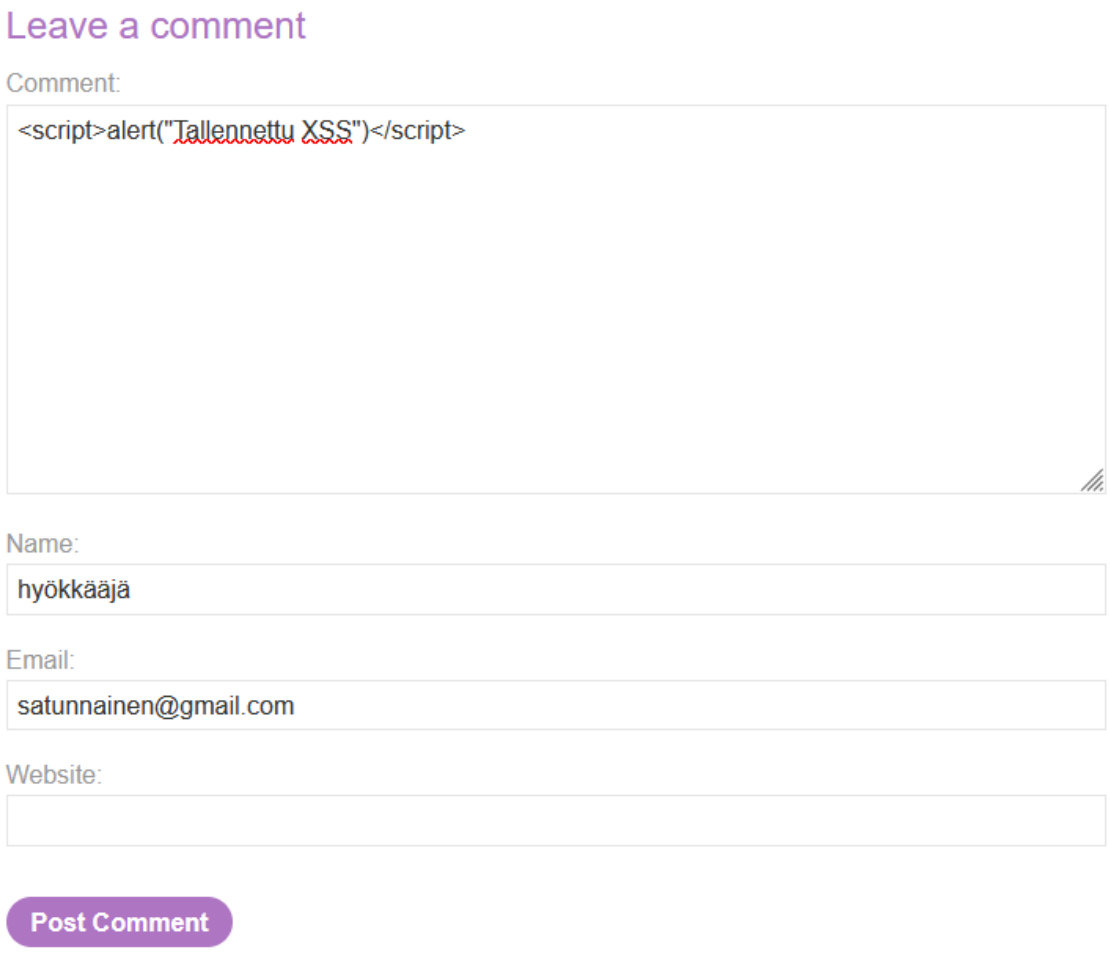
Kuva 2: Hyökkäyskoodi syötettynä hakukenttään

Kuvassa 2 tämä hyökkäyskoodi on syötettynä sivun hakukenttään. Hakukentässä nakyy vain osa koodista kerrallaan. Koodissa oleva `img`-tagi lisää sivulle uuden kuvan, jonka lähteenä on kuvan sijainnin sijaan numero ”1”. Tämä aiheuttaa virheen ohjelman suorituksessa, jolloin `img`-elementin `onerror`-attribuutissa määritelty JavaScript-koodi suoritetaan. Esimerkissä käytetään `window.open()`-funktiota, joka avaa uuden välilehden selaimessa. Funktion parametrinä annettu osoite määrittää uudelle välilehdelle avautuvan sivun osoitteen, joka tässä tapauksessa on hyökkääjän käyttämä `webhook.site`-niminen sivusto. Osoitteen lopussa oleva `?cookie=”+document.cookie` hakee `document`-olioa käyttäen uhrin evästeet ja liittää ne osoitteen loppuun hakuparametrinä. Kun selain pyytää tässä osoitteessa olevan sivuston sisällön, se lähettää myös samalla pyynnössä mukana hakuparametrissä olevat evästeet.

Kuva 3: Uhrin evästeet hyökkääjän webhook.site-sivustolla

Kuvassa 3 hakuparametrinä liitettyt evästeet näkyvät webhook.site-sivustolla. Kuvassa korostetussa `token`-evästeessä on uhrin instunnon yksilöivä avain. Asettamalla tämän omiin evästeisiinsä hyökkääjä voi toimia sivustolla käyttäen uhrin käyttäjää ilman, että hyökkääjän tarvitsisi kirjautua sisään käyttäjätunnuksilla. Näin hyökkääjä voi siis kaapata käyttäjän istunnon XSS-haavoittuvuutta hyödyntävällä hyökkäyksellä.

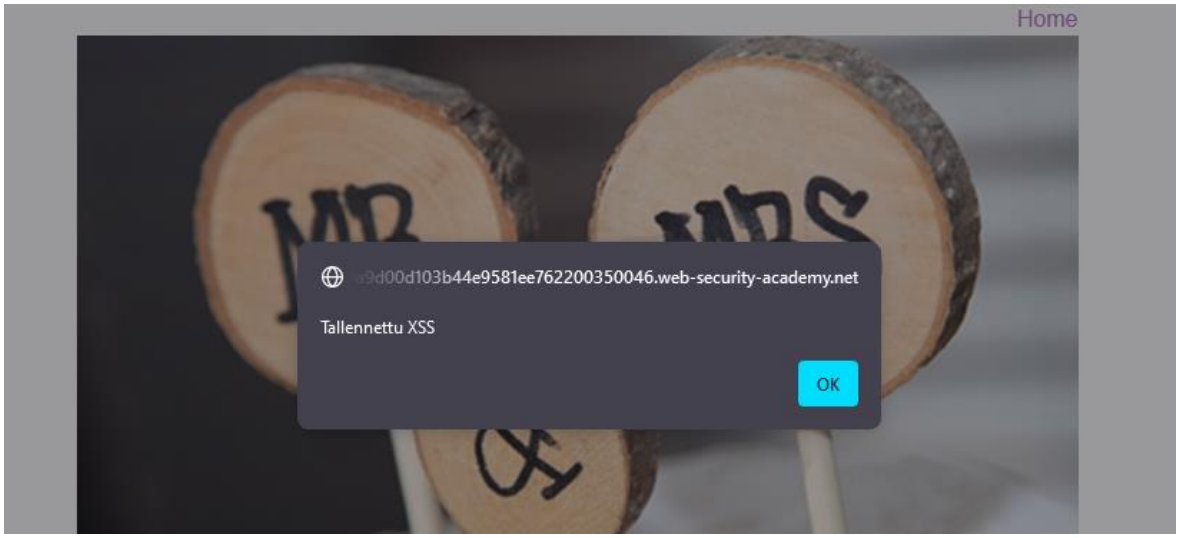
Tallennetun XSS-haavoittuvuuden esittämiseen käytetään PortSwiggerin XSS-harjoitusympäristöä.[2] Tämän harjoituksen sivustoon on tarkoituksella tehty tallennettu XSS-haavoittuvuus. Harjoitussivusto esittää blogisivustoa, jolla voi lukea blogikirjoituksia ja jättää niihin kommentteja. Haavoittuvaa kommentointiominaisuutta voidaan hyödyntää hyökkäyksessä. Sivun alaosassa on neljä tekstikenttää ja nappi, joka julkaisee kommentin sivulle. Kommentin tekstisisällölle tarkoitettuun Comment-kenttään syötettyä sisältöä ei suodateta millään tavalla, joten sitä voidaan käyttää halutun hyökkäyskoodin injektoimiseen sivustolle. Muilla syötekentillä ei ole juurikaan väliä, joten ne voi täyttää satunnaisesti. Comment-kenttään syötetään seuraava koodi: `<script>alert(”Tallennettu XSS”)</script>`



Kuva 4: Sivun comment-kenttä, johon on syötetty hyökkäyskoodi injektoimista varten

[2] Tallennetun XSS-haavoittuvuuden harjoitusympäristö: https://portswigger.net/web-security/cross-site-scripting/stored/lab-html-context-nothing-encoded

Kuvassa 4 syötekentät on täytetty ja Comment-kentään on sijoitettu hyökkäyskoodi valmiiksi injektoimiseen. Name- ja Email-kenttiä ei voi jättää tyhjäksi, joten nekin on täytetty esimerkkitiedoilla. Website-kenttään ei tarvitse syöttää mitään tällä sivulla, joten se on jätetty tyhjäksi. Kun koodi on injektoitu sivulle, se avaa sivulle ponnahdusikkunan, jossa lukee teksti ”Tallennettu XSS”. Tämän havainnollistavan koodin sijaan sivulle voitaisiin tietenkin injektoida jotakin vaarallista koodia.



Kuva 5: Hyökkäyskoodi suoriutuu ja sivulle avautuu ilmoitus

Sivun ulkoasu injektoinnin jälkeen näkyy kuvassa 5. Hyökkäyskoodin suorittaminen avaa sivulle ilmoitusikkunan, jossa lukee hyökkäyskoodissa ollut teksti. Kuvassa 6 esitetään osa sivun HTML-dokumenttia, jossa nähdään sivulle injektoitu koodi. Yksi sivun `kommentti`-luokan osioista sisältää hyökkäyskoodin. Näiden `<p></p>` -tagien välissä kuuluisi normaalisti olla kommentin tekstisisältö, mutta sivuston haavoittuvuutta käyttäen siihen pystyttiin injektoimaan mielivaltaista hyökkäyskoodia.

```
<section class="comment">
  <p>
    <img class="avatar" src="/resources/images/avatarDefault.svg">
    hyÃ¶kkÃ¤Ã¤jÃ¤ | 06 February 2024
  </p>
  <p>
    <script>alert("Tallennettu XSS")</script>
  </p>
  <p></p>
</section>
<hr>
```

Kuva 6: Osa sivun HTML dokumenttia jossa näkyy injektoitu koodi

Toisin kuin heijastetun XSS:n tapauksessa, tämä hyökkäys ei vaadi linkin levittämistä käyttäjille. Injektoitu koodi on nyt tallennettu sivulle ja se suorittuu joka kerta, kun käyttäjä vierailee tällä sivulla. Sivuston pääsivun ulkoasu ei ole muuttunut eikä käyttäjä voi tietää injektoidun koodin olemassaolosta ennen sivun avaamista. Juuri tämä olennainen ero tekee tallennetusta XSS:stä erityisen vakavan haavoittuvuuden.

Myös **DOM XSS-haavoittuvuuden** esimerkissä hyödynnetään PortSwiggerin luomaa harjoitusympäristöä.[3] Kuvassa 7 näkyy blogisivuston hakusivua mallintava harjoitussivusto. Sivulla on hakukenttä, jota käyttäen käyttäjä voi syöttää hakusanan ja hakea blogeja sivustolta. Tällaisissa toiminnoissa saattaa hyvinkin olla mahdollisuus XSS-haavoittuvuudelle.

Kuva 7: Harjoitussivu ja hakutoiminto

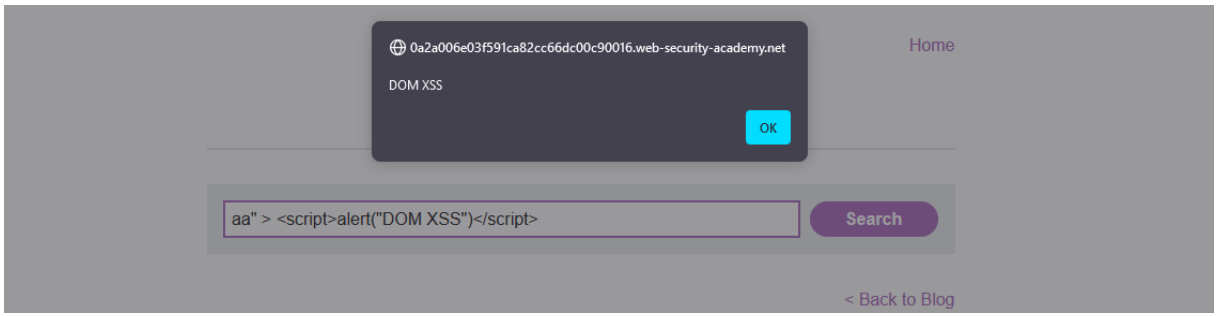
DOM XSS muistuttaa ulkoisesti heijastettua XSS:ää, mutta sen toimintaperiaate on hyvin erilainen. Tässä tapauksessa haavoittuvuus sijaitsee sivuston hakutoimintoon liittyvässä funktiossa, joka sijaitsee selaimen käsittelemässä käyttäjäpuolen koodissa.

```
<script>
    function trackSearch(query) {
        document.write('<img src="/resources/images/tracker.gif?searchTerms='+query+'">');
    }
    var query = (new URLSearchParams(window.location.search)).get('search');
    if(query) {
        trackSearch(query);
    }
</script>
```

Kuva 8: Haavoittuvainen JavaScript-funktio sivun HTML-dokumentissa

[3] DOM-XSS haavoittuvuuden harjoitussivusto: https://portswigger.net/web-security/cross-site-scripting/dom-based/lab-document-write-sink

Kuvassa 8 esitetään osa sivun HTML-dokumenttia, jossa näkyy haavoittuvainen funktio. Funktio lisää sivulle uuden `img`-elementin, jonka src-attribuuttiin lisätään funktiolle annettu `query`-muuttuja. Tämä `query`-muuttuja saadaan `window.location.search` lähteestä eli sivun URL-osoitteessa olevasta hakutermistä. Hakutermi on tietenkin käyttäjän hallittavissa, joten tätä kautta voidaan mahdollisesti injektoida omaa koodia sivulle. Injektio tapahtuu HTML-tagin attribuutin sisälle, josta täytyy päästä ulos, jos halutaan injektoida oma `script`-elementti. Tässä tapauksessa tämä tapahtuu helposti lisäämällä merkit `”>` injektoitavan koodin eteen. Tämä sulkee `img`-tagin ja injektoitu koodi luodaan uudeksi HTML-tagiksi sivulle. Injektioon käytetään seuraavaa koodia: `”> <script>alert(”DOM XSS”)</script>` .



Kuva 9: Hyökkäyskoodi suoriutuu ja sivulle avautuu ilmoitus

Tämä saa jälleen aikaan ponnahdusikkunan ladattaessa URL-osoitteen osoittama sivu, mikä voidaan nähdä kuvassa 9. Ikkunan ilmestyminen on todistus siitä, että sivulla on XSS-haavoittuvuus ja sitä käyttäen onnistuttiin injektoimaan mielivaltaista koodia sivulle. Kuvassa 10 injektoitu koodi näkyy myös sivun HTML-dokumentissa. Voi huomata, että on onnistuttu pääsemään ulos `img`-tagin kontekstista ja injektoitu oma `script`-elementti dokumenttiin. Hakukenttään syötetyn hakusananhan oli alunperin tarkoitus olla osa `img`-tagin `src`-attribuutissa olevaa osoitetta.

```
<img src="/resources/images/tracker.gif?searchTerms=aa">
<script>alert("DOM XSS")</script>
```

Kuva 10: Osa sivun HTML-dokumenttia, jossa näkyy injektoitu koodi alemmalla rivillä

Käyttämällä jotakin haitallista koodia tässä injektiossa ja levittämällä linkkiä käyttäjille voidaan suorittaa DOM-pohjainen XSS-hyökkäys käyttäjiä vastaan.

Nämä esimerkit ovat hyvin yksinkertaisia, eivätkä nykyajan web-sivustot usein ole näin haavoittuvaisia. Useasti saman lopputuloksen aikaansaamiseen vaaditaan huomattavasti

monimutkaisempia ja mielikuvituksellisempia menetelmiä. Tästä huolimatta edelleen löytyy sivustoja, jotka ovat esimerkkien tavoin täysin suojaamattomia XSS-hyökkäyksiltä. Tosin nämä sivustot todennäköisesti eivät ole ammattilaisten suunnittelemia.

3 XSS-haavoittuvuuksien etsiminen sovelluksesta

Sovellukseen kohdistuvien mahdollisten cross site scripting -hyökkäyksien ennaltaehkäisyn kannalta on tärkeää havaita haavoittuvuudet ennen hyökkääjiä. Optimaalista olisi tietysti havaita haavoittuvuudet jo ennen sovelluksen tai sen uuden version julkaisua, jotta ne voidaan korjata ennen kuin hyökkääjät voivat päästä sovellukseen käsiksi.

Eri havaitsemistekniikat hyödyntävät erilaisia analyysimenetelmiä. Ne voidaan jakaa analyysimenetelmänsä mukaan kolmeen kategoriaan, jotka ovat staattinen analyysi, dynaaminen analyysi ja näiden kahden välimuoto: hybridianalyysi. Staattiset analyysimenetelmät pääosin etsivät potentiaalisia haavoittuvuuksia analysoimalla web-sovelluksen lähdekoodia. Nämä menetelmät johtavat kuitenkin usein vääriin positiivisiin tuloksiin. Lisäksi lähdekoodi ei ole aina saatavilla analysoitavaksi. Dynaamiset analyysimenetelmät taas etsivät haavoittuvuuksia injektoimalla dataa web-sivustolle ja seuraamalla, saako syöte aikaan hyökkäyksen. Nämä menetelmät puolestaan jatkuvasti aliarvioivat haavoittuvuuksien määrää, sillä ne eivät kykene kattamaan kaikkia mahdollisia tapauksia. Hybridianalyysiä käyttävät menetelmät yhdistävät kahden aikaisemman menetelmän piirteitä haavoittuvuuksien löytämiseksi. (Liu ja muut 2019.)

Koneoppimiseen perustuvien menetelmien tehokkuutta arvioidaan tavallisesti mallin sisäisen tarkkuuden (engl. precision), saannin (engl. recall) ja F1-tarkkuuden (engl. F1 score) perusteella. Haavoittuvuuksien havaitsemisen kontekstissa mallin sisäisellä tarkkuudella tarkoitetaan osuutta havaituista haavoittuvuuksista, jotka ovat oikeasti haavoittuvuuksia eivätkä vääriä positiivisia. Saannilla tarkoitetaan sitä osuutta kaikista aineistossa olevista haavoittuvuuksista, jotka malli havaitsi. F1-tarkkuus on sisäisen tarkkuuden ja saannin harmoninen keskiarvo. Kaikki kolme arvoa saavat siis jonkin arvon nollan ja yhden väliltä. Malli, jonka F1-tarkkuus haavoittuvuuksien havaitsemisessa on 1 olisi täydellinen, sillä se havaitsisi jokaisen aineistossa olevan haavoittuvuuden eikä tuottaisi yhtäkään väärää positiivista.

Koneoppimista hyödyntämättömien menetelmien tehokkuutta arvioidaan tavallisesti ulkoisen tarkkuuden (engl. accuracy), väärien positiivisten osuuden (engl. false-positive rate) ja havaittujen haavoittuvuuksien määrän perusteella. Näissä tapauksissa ulkoisella tarkkuudella tarkoitetaan menetelmän oikein haavoittuvaksi tai ei-haavoittuvaksi luokittelemien tapausten suhdetta kaikkiin tapauksiin. Väärien positiivisten osuudella tarkoitetaan osuutta kaikista ei-

haavoittuvista tapauksista, jotka menetelmä luokitteli haavoittuvaksi. Havaittujen haavoittuvuuksien määrällä tarkoitetaan haavoittuvaksi tiedetyissä sovelluksissa havaittujen haavoittuvuuksien määrää. (Ayeni ja muut 2018.)

3.1 Staattinen analyysi

Staattisessa analyysissä käsitellään tavallisesti analysoitavan sovelluksen lähdekoodia. Mahdollisia haavoittuvuuksia voidaan löytää analysoimalla lähdekoodin rakennetta ja datavirtaa. (Li ja muut 2020.) Datavirran analyysissä selvitetään mitä arvoja mikäkin muuttuja voi saada ohjelman suorituksen aikana ja miten nämä arvot leviävät ohjelman sisällä sekä missä ohjelman osissa niitä käytetään. Staattisen analyysin vahvuus verrattuna muihin kategorioihin on sen nopeus, mutta se usein merkitsee haavoittuvaksi myös osia lähdekoodista, jotka eivät todellisuudessa ole haavoittuvaisia (Li ja muut 2020).

Li ja muut (2020) esittävät työssään menetelmän XSS-haavoittuvuuksien havaitsemiseen PHP-koodikielellä kirjoitetussa lähdekoodissa. Menetelmä perustuu PHP-koodin jäsentämiseen (engl. parsing) ja koneoppimista hyödyntävään algoritmiin. Li ja muut analysoivat lähdekoodin operaatioiden sarjaa (engl. operation sequence). Tämän pohjalta he suunnittelivat koneoppimismallin, jonka avulla on mahdollista selvittää, miten ohjelman koodi käsittelee käyttäjän syötteitä ja havaita siinä haavoittuvuuksia. Menetelmän saavuttama F1-tarkkuus oli 0,92, mikä on melko hyvä tulos. Menetelmä löytää suuren osan haavoittuvuuksista, eikä tee huomattavia virheitä havaitsemisessa.

Myös Zhou ja muut (2020) hyödyntävät koneoppimista haavoittuvuuksien etsimiseen web-sovellusten lähdekoodista. Työssä esitetty Vulnerability Hunter (VulHunter) -niminen menetelmä perustuu syväoppimiseen ja tavukoodin (engl. bytecode) hyödyntämiseen. Tavukoodi on lähdekoodista käännetty käskysarja (engl. instruction set), joka on tarkoitettu tulkkaajan (engl. interpreter) suoritettavaksi. Tavukoodi on ikään kuin väliversio lähdekoodin ja konekielen välillä, jonka tarkoitus on mahdollistaa saman koodin suorittaminen eri alustoilla. Tavukoodi usein suoritetaan virtuaalikoneessa, jossa se käännetään konekielelle. VulHunter käyttää tätä tavukoodia neuroverkon syötteenä ja laskee analysoitavan sovelluksen ja mallihaavoittuvuuksien samankaltaisuutta. Tämän perusteella arvioidaan, onko ohjelma haavoittuvainen. Menetelmän tehokkuutta arvioitiin SQL-injektio- ja XSS-haavoittuvuuksien havaitsemisessa PHP-ohjelmissa. VulHunter saavutti XSS-haavoittuvuuksien havaitsemisessa 0,95 F1-tarkkuuden. Käytännön testauksessa menetelmää käyttäen löydettiin viisi uudenlaista haavoittuvuutta.

Tuoreemmassa tutkimuksessa Liu ja muut (2023) esittävät menetelmän XSS-haavoittuvuuksien havaitsemiseen käyttäjäpuolen JavaScript-koodissa. Menetelmä perustuu neuroverkkojen käyttöön. Neuroverkot analysoivat koodin ja poimivat siitä haavoittuvuuden kannalta olennaiset piirteet (engl. feature). Tämän tiedon avulla voidaan havaita haavoittuvuuksia käyttäjäpuolen koodissa. Olennainen ero muihin neuroverkkoja hyödyntäviin menetelmiin on sekä graafien että merkkijonoiksi muutetun koodin (engl. code string) käyttäminen piirteiden poimimiseen (engl. feature extraction). Piirteiden poimimisessa graafeilla (Graph Feature Extraction) lähdekoodi esitetään graafirakenteena. Tätä graafia käsittelemällä ja analysoimalla neuroverkko poimii olennaiset piirteet lähdekoodista. Merkkijonoiksi muutetun koodin (Code String Feature Extraction) tapauksessa taas lähdekoodista erotellaan olennaiset funktiot ja attribuutit, jotka muutetaan merkkijonoiksi. Tätä käsittelemällä neuroverkko poimii olennaiset piirteet lähdekoodista. Menetelmien yhdistäminen tuottaa parempia tuloksia. Menetelmä suoriutui vertailtavia menetelmiä paremmin saavuttaen 0,997 F1-tarkkuuden, 0,997 sisäisen tarkkuuden ja 0,997 saannin.

3.2 Dynaaminen analyysi

Dynaamisessa analyysissä keskitytään ohjelman suorituksen aikana saatavaan tietoon. Menetelmät havaitsevat XSS-haavoittuvuuksia lähettämällä HTTP-pyyntöjä web-sivuston palvelimelle ja tulkitsemalla palvelimen vastauksia. Dynaamisten analyysimenetelmien olennainen vahvuus on niiden toimivuus ilman sovelluksen lähdekoodia. Ne myös tuottavat paljon vähemmän vääriä hälytyksiä verrattuna staattisiin analyysimenetelmiin. Dynaaminen analyysi on kuitenkin huomattavasti hitaampaa, sillä testattavien hyökkäysten määrän kasvaessa myös testaukseen kuluva aika lisääntyy. Tämä aikavaatimus voi tehdä menetelmistä käyttökelvottomia käytännössä. Menetelmiltä saattaa myös jäädä haavoittuvuuksia havaitsematta, sillä testauksessa ei ole mahdollista käydä läpi kaikkia mahdollisia hyökkäysvaihtoehtoja. (Liu ja muut 2019.)

Ayeni ja muut (2018) esittävät CrawlerXSS-nimisen menetelmän XSS-haavoittuvuuksien havaitsemiseen web-sovelluksissa. Menetelmä perustuu sumean logiikan (engl. fuzzy logic) hyödyntämiseen DOM-XSS-haavoittuvuuksien havaitsemisessa. Sumeassa logiikassa totuusarvo voi olla diskreetin toden ja epätoden tai 1 ja 0 lisäksi myös joku niiden välissä olevista reaaliluvuista. Täten mahdollisia totuusarvoja on kahden sijaan ääretön määrä. Näitä numeerisia arvoja käytetään ongelmien ratkaisun johtamiseen. Sumea logiikka on ikään kuin klassisen logiikan laajennus, jonka avulla voidaan käsitellä ongelmia, joissa kuvataan

tietämystä (engl. knowledge representation) epävarmassa ja epätarkassa ympäristössä. Menetelmä erottaa web-sivustolta elementit, joissa voi olla DOM-XSS-haavoittuvuuksia sekä niihin liittyvät attribuutit. Nämä syötetään sumealla logiikalla toimivaan päättelyjärjestelmään, joka havaitsee mahdolliset haavoittuvuudet. Menetelmän ulkoinen tarkkuus haavoittuvuuksien havaitsemisessa on 95 % ja väärien positiivisten osuus 0,99 %. Menetelmää käyttäen löydettiin haavoittuvuuksia kaikista testatuista web-sivustoista. Menetelmä suoriutui huomattavasti paremmin kuin aikaisemmin julkaistut menetelmät, joihin sitä verrattiin.

Liu ja muut (2022) esittävät geneettiseen algoritmiin perustuvan fuzz-testausmallin XSS-haavoittuvuuksien havaitsemiseen. Fuzz-testaus on ohjelmistotestausmenetelmä, jossa sovellukselle annetaan vääränlaisia, arvaamattomia tai satunnaisia syötteitä. Tavoitteena on löytää sovelluksesta haavoittuvuuksia tai muita ongelmia, jotka johtavat esimerkiksi sovelluksen kaatumiseen. Liun ja muiden (2022) menetelmä kykenee generoimaan testisyötteitä dynaamisesti erilaisille syötepisteille web-sovelluksessa. Yksittäisen syötteen kelpoisuus arvioidaan sovelluksen lähettämän vastauksen perusteella. Geneettinen algoritmi valitsee ja mutatoi syötteet, siten että syötteet, joilla on parempi kelpoisuus, valitaan todennäköisemmin. Tämän jälkeen testaus suoritetaan uudelleen seuraavalla sukupolvella. Prosessia jatketaan, kunnes löydetään haavoittuvuus tai saavutetaan piste, jossa syötteiden kelpoisuus ei enää parane. Testien perusteella tämä oli yleensä noin 30. sukupolven kohdalla, joten kierrosten ylärajaksi asetettiin 30. Testauksessa menetelmän ulkoinen tarkkuus oli 1 ja saanti 0,815. Menetelmä ei siis tuottanut yhtäkään väärää positiivista. Tämä on yleistä menetelmissä, jotka arvioivat haavoittuvuutta sen perusteella saadaanko jonkinlainen hyökkäyskoodi suoritettua. Menetelmän keskimääräinen suoritusaika oli huomattavasti lyhyempi kuin verrattavilla menetelmillä, mutta sen aikahajonta oli suurempaa. Keskimääräinen suoritusaika oli 75,5 sekuntia, mutta pahimmillaan suorituksessa voi mennä 160 sekuntia.

3.3 Hybridianalyysi

Hybridianalyysimenetelmissä yhdistyy sekä staattisen että dynaamisen analyysin vahvuudet. Menetelmät pystyvät kartoittamaan sovelluksen rakenteen ja toiminnan tarkasti lähdekoodin avulla ja lisäksi niiden väärien positiivisten osuus on hyvin alhainen. Tavallisesti hybridianalyysissä käytetään staattista analyysia mahdollisten haavoittuvuuksien havaitsemiseen ja korkean havaitsemisnopeuden saavuttamiseen. Dynaamista analyysiä hyödynnetään tämän jälkeen havaittujen haavoittuvuuksien varmistamiseen. Tällä

yhdistelmällä ei kuitenkaan pystytä paikkaamaan kaikkia staattisen analyysin puutteita. Osa menetelmistä on yhteensopivia vain yhden ohjelmointikielen kanssa, mikä rajoittaa niiden käytettävyyttä. (Liu ja muut 2019.)

Ahmed ja Ali (2016) esittävät staattista ja dynaamista analyysiä hyödyntävän menetelmän XSS-haavoittuvuuksien havaitsemiseen. Menetelmässä analysoidaan lähdekoodia ja havaitaan siinä mahdollisia haavoittuvuuksia taint-analyysin avulla. Havaittuja haavoittuvuuksia testataan generoitujen esimerkkihyökkäysten avulla. Esimerkkihyökkäysten generoimiseen käytettään geneettistä algoritmia. Geneettisen algoritmin syötteenä käytettään tietokantaa erilaisista XSS-hyökkäyksistä. Menetelmä kykenee myös ottamaan uudenlaiset hyökkäykset huomioon, jos ne lisätään käytettävään tietokantaan. Menetelmä voidaan käyttää ainoistaan PHP-kielellä kirjoitettujen web-sovellusten analysoimiseen. Lisäksi menetelmää testattiin vain pienimuotoisissa sovelluksissa, joten sen toimivuudesta laajempien ja monimutkaisempien sovellusten analysoimisessa ei voida olla varmoja. Ahmed ja Alin mukaan työn kirjoitushetkellä ei ollut löydettävissä verrattavia menetelmiä. Tästä syystä menetelmää verrattiin menetelmään, jossa tietokannasta valittiin satunnaisesti hyökkäyksiä testattavaksi. Geneettistä algoritmia käyttävä menetelmä toimi tehokkaammin kuin satunnainen valinta.

Song ja muut (2023) esittävät menetelmän heijastettujen ja tallennettujen XSS-haavoittuvuuksien havaitsemiseen Java-kielellä kirjoitetuissa web-sovelluksissa. Ensin staattisella analyysillä havaitaan osat sovelluksen komponenteissa, joita voidaan käyttää injektoimiseen. Tämän jälkeen generoidaan testisyötteet, joita käytettään havaittujen pisteiden fuzz-testaamiseen. Tämän dynaamisen osuuden yhteydessä hyödynnetään vahvistusoppimista (engl. reinforcement learning) testisyötteiden optimoimiseen ja havaitsemisnopeuden parantamiseen. Vahvistusoppiminen on koneoppimisen osa-alue, jossa agentti pyrkii löytämään tehokkaimman mahdollisen tavan toimia ympäristössä. Agentti toimii ympäristön tilan mukaan ja saa toimestaan positiivisen tai negatiivisen palkinnon riippuen sen tehokkuudesta. Palkinnon perusteella agentti muuttaa toimintaansa ja yrittää uudelleen, kunnes se löytää tehokkaimman toimintatavan. Tässä menetelmässä agentti palkitaan positiivisesti, jos XSS-haavoittuvuus havaitaan ja negatiivisesti jos haavoittuvuutta ei löydetty tai jos samaa testisyötettä on jo käytetty viimeisen kymmenen kierroksen aikana. Menetelmä testattiin kahdella web-sivustolla, joista molemmista löydettiin kaikki haavoittuvuudet ilman yhtäkään väärää positiivista. Eli saavutettu sisäinen tarkkuus ja saanti sekä täten myös F1-tarkkuus olivat kaikki 1.

Taulukko 1. Havaitsemismenetelmät tiivistettynä

Tutkimus	**Analysoinnin kategoria**	**Keskeiset piirteet**
Li ja muut (2020)	Staattinen	PHP-koodin jäsentäminen ja analysointi hyödyntäen koneoppimista
Zhou ja muut (2020)	Staattinen	Tavukoodin analysointi syväoppivalla neuroverkolla
Liu ja muut (2023)	Staattinen	Piirteiden poimiminen käyttäen graafeja ja merkkijonoiksi muunnettua lähdekoodia
Ayeni ja muut (2018)	Dynaaminen	Haavoittuvuuksien havaitseminen käyttäen sumeaa logiikkaa
Liu ja muut (2022)	Dynaaminen	Fuzz-testaus käyttäen geneettistä algoritmia
Ahmed ja Ali (2016)	Hybridi	Taint-analyysi ja hyökkäysten generointi käyttäen geneettistä algoritmia
Song ja muut (2023)	Hybridi	Staattisesti havaittujen haavoittuvuuksien testaaminen vahvistusoppimista hyödyntävällä fuzz-testauksella.

Taulukossa 1 listataan käsitellyt menetelmät XSS-haavoittuvuuksien havaitsemiseen. Menetelmät on listattu järjestyksessä menetelmän tyypin ja julkaisuajankohdan mukaan, mikä on myös sama järjestys kuin missä ne käsiteltiin tässä luvussa. Taulukosta 1 näkee selvästi, miten käsitellyt menetelmät jakautuvat kategorioihin analysointitavan mukaan sekä olennaiset piirteet menetelmien toteutuksessa.

4 Pohdinta

XSS-haavoittuvuudet ovat hyvin keskeinen ja kasvava ongelma nykyhetkellä, joten tämän ongelman ratkaisuun pyrkivää tutkimusta on myös runsaasti. Näitä julkaisuja läpikäydessä heräsi useita erilaisia mietteitä, joita tässä luvussa käsitellään. Menetelmiä haavoittuvuuksien havaitsemiseen käsiteltiin kategorioittain niiden käyttämän analyysimenetelmän mukaan, mutta tässä lähestymistavassa on ilmeinen ongelma. Menetelmien kategorisointi johonkin näistä luokista on melko usein haastavaa. Vain jotkut menetelmistä asettuvat selvästi yhteen tiettyyn kategoriaan. Erityisen mutkikasta on määritellä, kuuluuko jokin menetelmä hybridianalyysin piiriin vai käytettiinkö menetelmässä esimerkiksi pelkästään dynaamista analyysiä.

Lisäksi tämän aihepiirin julkaisuissa käytetyt menetelmät tehokkuuden mittaamiseen vaihtelevat huomattavasti. Tämä on huomattavissa jo pelkästään käsitellyissä seitsemässä julkaisuissa. Selkeä ero on esimerkiksi perinteisempien menetelmien ja koneoppimista hyödyntävien menetelmien välillä. Kuitenkin myös näiden kategorioiden sisällä käytetyissä käytännöissä on eroja. Joissain julkaisuissa ilmoitetaan sisäinen ja ulkoinen tarkkuus sekä saanti ja F1-tarkkuus. Toisissa taas pelkästään toinen sisäisestä tai ulkoisesta tarkkuudesta. Eroavat arviointikäytännöt hankaloittavat menetelmien vertailua. Arviointikäytäntöjen yhtenäistäminen helpottaisi vertailua huomattavasti ja selkeyttäisi tutkimuksen tilanteen hahmottamista.

Analysointitavat eivät ole kaikki yhtä yleisiä ja käytettyjä. Staattiseen analyysiin luokiteltavia menetelmiä oli selvästi enemmän kuin esimerkiksi niitä, joissa käytettiin hybridianalyysiä. Tämä saattaa johtua siitä, että staattisen ja dynaamisen analyysin yhdistäminen on monimutkaisempaa kuin vain yhden analysointitavan käyttäminen.

Näiden seikkojen lisäksi huomioitavaa on myös termistön epäsäännöllinen käyttö. Erityisesti termien XSS-haavoittuvuus ja XSS-hyökkäys sekoittaminen on huomattavaa. Joissain tilanteissa käytettiin termiä XSS-hyökkäys, vaikka todellisuudessa käsiteltiin haavoittuvuuksien havaitsemista. Toisaalta taas joissain tilanteissa käytettiin termiä XSS-haavoittuvuus, vaikka todellisuudessa käsiteltiin web-sovelluksiin kohdistuvien hyökkäysten reaaliaikaista havaitsemista ja torjumista. Säännöllisyys ja yhteiset käytänteet termien käytössä olisivat varmasti tehokkaan tutkimuksen ja selvemmän ymmärryksen eduksi.

Käsiteltyjen julkaisujen tuloksista on huomattavissa useita mielenkiintoisia ja positiivisia seikkoja XSS-haavoittuvuuksien havaitsemismenetelmien kehityksestä.

Havaitsemismenetelmien tehokkuus on korkealla tasolla ja menetelmät ovat kehittyneet huomattavasti viimeisen kymmenen vuoden aikana. Staattisen analyysin osalta Li ja muut (2020) saavuttivat 0,92 F1-tarkkuuden jo vuonna 2020. Tämä on jo hyvin tehokas menetelmä, mutta vuonna 2023 Liu ja muut esittivät menetelmän, joka kykenee jopa 0,97 F1-tarkkuuteen. Dynaamista analyysiä käyttäen Liu ja muut (2022) kykenivät kehittämään menetelmän, joka saavutti 1 sisäisen tarkkuuden, mutta menetelmän saanti oli vain 0,815. Kuitenkin alustavien kokeiden perusteella tehokkaimpana on Song ja muiden (2023) esittämä menetelmä, joka saavuttaa täydellisen 1 F1-tarkkuuden. Kaikki nämä tulokset ovat tietenkin saavutettu kokeissa käytetyllä aineistolla. Menetelmät eivät välttämättä kykene samaan tehokkuuteen kaikissa web-sovelluksissa, eivätkä välttämättä havaitse kaikista monimutkaisimpia haavoittuvuuksia.

Ilmeisen selvänä rajoitteena useissa havaitsemismenetelmissä on kyky toimia vain tietyillä ohjelmointikielillä kehitettyjen sovellusten kanssa tai mahdollisuus havaita vain tietynlaisia XSS-haavoittuvuuksia. Menetelmän käytännöllisyys heikkenee huomattavasti, jos sitä voidaan käyttää vain PHP-kielellä kehitetyissä sovelluksissa, kuten Zhou ja muiden (2020) menetelmässä. Samoin saavutettu suoja on hyvin rajattu, jos kyetään havaitsemaan vain esimerkiksi DOM-XSS-haavoittuvuuksia, kuten Ayeni ja muiden (2018) menetelmässä. Näitä seikkoja pyritään kuitenkin varmasti korjaamaan tulevissa töissä.

Selvästi kasvava ilmiö XSS-haavoittuvuuksien havaitsemismenetelmissä on tekoälyä, kuten kone- ja syväoppimista hyödyntävät menetelmät. Hyvin monet uudemmista havaitsemismenetelmistä käyttävät tekoälyä jossain määrin haavoittuvuuksien havaitsemiseen. Esimerkkejä tällaisista menetelmistä ovat esimerkiksi Zhou ja muut (2020), Liu ja muut (2023) sekä Song ja muut (2023). Tekoälyn käyttö on yleistynyt monella alalla, joten ei ole yllättävää, että myös XSS-haavoittuvuuksien havaitsemisessa pyrittäisiin hyödyntämään sitä. Tekoälyn käyttö tulee todennäköisesti kasvamaan tulevaisuudessa.

Mielenkiintoista havaitsemismenetelmissä on lähestymistapojen monipuolisuus. Tutkimuksessa ei selvästikään ole juututtu yhteenkään parhaaksi katsottuun tapaan havaita haavoittuvuuksia. Uusissa tutkimuksissa pyritään löytämään uusia näkökulmia ja ratkaisuja XSS-haavoittuvuuksien tehokkaampaan ja tarkempaa havaitsemiseen web-sovelluksissa. Tämä on hyvin loogista, sillä löytääkseen uudenlaisia haavoittuvuuksia on ajateltava uudella tavalla ja lähestyä ongelmaa eri tavalla kuin aikaisemmin.

5 Yhteenveto

Cross-site Scripting haavoittuvuudet ovat yksi yleisimmistä web-sovellusten haavoittuvuuksista ja täten hyvin tärkeä aihe tietoturvallisuuden alalla. Tässä tutkielmassa pyrittiin selvittämään olemassa olevia menetelmiä, joilla XSS-haavoittuvuuksia voidaan havaita web-sovelluksissa. Haavoittuvuuksien havaitseminen on olennaista niiden korjaamisen ja sovelluksen turvallisuuden kannalta.

Tutkimuskysymykseen vastattiin luvuissa 3 ja 4 käymällä läpi haavoittuvuuksien havaitsemismenetelmiä. XSS-haavoittuvuuksia voidaan havaita ohjelmilla tai koneoppimismalleilla, jotka tunnistavat mahdollisia haavoittuvuuksia web-sovelluksissa. Menetelmät havaitsevat haavoittuvuuksia analysoimalla sovelluksen lähdekoodia tai kohdistamalla testihyökkäyksiä web-sovellukseen. Jotkut menetelmät myös yhdistävät näitä kahta analyysitapaa paikatakseen niiden heikkouksia. Menetelmät eroavat näiden kategorioiden sisällä toisistaan lähinnä niiden tavassa tunnistaa haavoittuvuuksia jäsennetyssä lähdekoodissa tai tavassa generoida ja kehittää esimerkkihyökkäyksiä.

Uusimmat menetelmät ovat hyvin tehokkaita ja alustavien kokeiden perusteella kykenevät löytämään kaikki haavoittuvuudet testiaineistossa ilman yhtäkään väärää positiivista. Menetelmien toimivuutta käytännössä tulisi tutkia laajemmin, jotta voitaisiin selvittää kuinka hyvin ne soveltuvat todelliseen käyttöön oikeissa web-sovelluksissa. Tekoälyn soveltaminen XSS-haavoittuvuuksien havaitsemisessa on kasvava ilmiö, ja sitä hyödyntävät menetelmät ovat saavuttaneet huomattavia tuloksia. Tekoälyn mahdollistamia menetelmiä tulisi tutkia tulevaisuudessakin. Usean tämänhetkisen menetelmän ongelmana on rajoitukset ympäristöjen ja sovelluksen kehittämiseen käytetyn ohjelmointikielen kanssa. Olemassa olevia menetelmiä tulisi laajentaa kattamaan useampia ympäristöjä ja ohjelmointikieliä. Ratkaistavana ongelmana on myös joidenkin menetelmien kyky havaita vain tietynlaisia XSS-haavoittuvuuksia. Menetelmät ovat kuitenkin kehittyneet nopeasti vuoden 2015 jälkeen ja tämä kehitys tulee todennäköisesti jatkumaan tulevaisuudessakin.

Lähteet

Ahmed, M. A., & Ali, F. (2016). Multiple-path testing for cross site scripting using genetic algorithms. Journal of Systems Architecture, 64, 50–62. https://doi.org/10.1016/j.sysarc.2015.11.001

Ayeni, B. K., Sahalu, J. B., & Adeyanju, K. R. (2018). Detecting Cross-Site Scripting in Web Applications Using Fuzzy Inference System. Journal of Computer Networks and Communications, 2018, 1–10. https://doi.org/10.1155/2018/8159548

Choi, Y. B., Loo, Y. L., & LaCroix, K. (2019). Cookies and sessions: A study of what they are, how they can be stolen and a discussion on security. International Journal of Advanced Computer Science & Applications, 10(1), 32–36. https://doi.org/10.14569/IJACSA.2019.0100104

Cui, Y., Cui, J., & Hu, J. (2020, Helmikuu). A survey on xss attack detection and prevention in web applications. In Proceedings of the 2020 12th International Conference on Machine Learning and Computing (pp. 443-449).

Grossman, Jeremiah. (2007). XSS attacks: cross-site scripting exploits and defense. Burlington, Mass: Syngress.

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., & Linkman, S. (2009). Systematic literature reviews in software engineering–a systematic literature review. Information and software technology, 51(1), 7-15.

Li, C., Wang, Y., Miao, C., & Huang, C. (2020). Cross-site scripting guardian: A static XSS detector based on data stream input-output association mining. Applied Sciences, 10(14), 4740-. https://doi.org/10.3390/app10144740

Liu, M., Zhang, B., Chen, W., & Zhang, X. (2019). A Survey of Exploitation and Detection Methods of XSS Vulnerabilities. IEEE Access, 7, 182004–182016. https://doi.org/10.1109/ACCESS.2019.2960449

Liu, Z., Fang, Y., Huang, C., & Xu, Y. (2022). GAXSS: Effective Payload Generation Method to Detect XSS Vulnerabilities Based on Genetic Algorithm. Security and Communication Networks, 2022, 1–15. https://doi.org/10.1155/2022/2031924

Liu, Z., Fang, Y., Huang, C., & Xu, Y. (2023). MFXSS: An effective XSS vulnerability detection method in JavaScript based on multi-feature model. Computers & Security, 124, 103015-. https://doi.org/10.1016/j.cose.2022.103015

OWASP. (2021). OWASP Top Ten 2021. https://owasp.org/Top10/

Rodríguez, G. E., Torres, J. G., Flores, P., & Benavides, D. E. (2020). Cross-site scripting (XSS) attacks and mitigation: A survey. Computer Networks (Amsterdam, Netherlands : 1999), 166, 106960-. https://doi.org/10.1016/j.comnet.2019.106960

Song, X., Zhang, R., Dong, Q., & Cui, B. (2023). Grey-Box Fuzzing Based on Reinforcement Learning for XSS Vulnerabilities. Applied Sciences, 13(4), 2482-. https://doi.org/10.3390/app13042482

Zhou, J., Luo, X., Shen, Q., & Xu, Z. (2020). VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode. In Information and Communications Security: 21st International Conference, ICICS 2019, pp. 199–218. Switzerland: Springer International Publishing AG. https://doi.org/10.1007/978-3-030-41579-2_12