

Ohjelmistojen haavoittuvuustestaus

Tuomas Tynkkynen

Aine

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Helsinki, 6. maaliskuuta 2013

Sisältö

1 Johdanto	1
2 Yleisiä tietoturvaongelmia	1
2.1 Muistiturvallisuus	2
3 Testausmenetelmät	4
3.1 Luokittelut	4
3.2 Testauksen kriteereitä	5
4 Staattinen analyysi (u hoholit)	5
4.1 Lint	5
5 Fuzzyaus	6
5.1 Testitapausten generointi	8
6 Yhteenvetot	8
Lähteet	9

vinnestdegssä
org. alihku,
pys (tai larenit)

1) ei yleisissä
alihkuja
2) ei 1 paragraf
(alihkuja
3) ei 1 virkhee
paragrafes
(... paitsi ns.
tsi hymytili/
korostussysteemi)
R 4) ei numerolle yh alkava virki tai
yleisn emen utte otsikko yh 63
virki virki (reihuzethispage{3em})
loitsii taittessesse pors)

1 Johdanto

Ohjelmistojen haavoittuvuukset ovat verrataen ikäviä: Tietoturva-aukoista aiheutuneesta ylimääräisestä työstä tietojärjestelmien ylläpitäjille sekä menetystä työajasta voi seurata suuria rahallisia tappioita, puhumattakaan mahdollisista henkilötietojen tai yrityssalaisuuksien vuotamisesta. Esimerkiksi Microsoftin IIS-palvelimen haavoittuvuuden avulla levinneestä Code Red-madosta arvioidaan aiheutuneen yhteensä noin 2,6 miljardin tappiot [MSC02]. Siksi on toivottavaa, että ohjelmiston tietoturvasta voidaan varmistua ennen ohjelmiston käyttöönottoa.

Tietoturvaongelmaa voi yrittää löytää käsityönä tutkimalla ohjelman lähdekoodia, mikä tietenkin on mahdollista vain ohjelmiston varsinaisille kehittäjille tai avoimen lähdekoodin ohjelmille. Lähdekoodin puuttuessa täytyy ensin ohjelmatiedosto takaisinkäantää disassembler-ohjelmalla symboliselle konekielelle ja tutkia ohjelmaa konekielitasolla. Kummassakin tapauksessa manuaalinen tutkiminen on työlästä ja aikaavievä, joten automatisoitua ratkaisu on paikallaan.

+ Luusse 2 :-)

2 Yleisiä tietoturvaongelmia

Tietoturvaavoittuvuuden vakavuuteen vaikuttaa keskeisesti aiheutuuko siitä *luottamusrajan* (trust boundary) ylitys. Esimerkiksi jos jokin palvelun kaatanan ohjelointivirheen voi laukaista pelkästään järjestelmän ylläpitäjä, ei kyseisellä virheellä ole suurta tietoturvamerkitystä. Lähes samaan lopputulokseenhan päädytään jos ylläpitäjä sammuttaa palvelun tavallisia keinoja käyttämällä. Sen sijaan jos vaikkapa Javascript-sovelmat selaimessa pääsee lukemaan käyttäjän tiedostoja, on tilanne vakavampi, koska web-sivujen katseleminen tarkoitus on olla turvallista ja selaimen siten kuuluu estää sovelmien pääsy tiedostojärjestelmään. Vastaavaan tapaan tavallisen käyttäjän pääsy suorittamaan ohjelmaa ylläpitäjän oikeuksilla tai käyttötietelmätilassa

rikoo luottamusrajan.

2.1 Muistiturvallisuus

Ohjelointikieli vaikuttaa ratkaisevasti siihen miten laajoja seurausia ohjelointivirheillä voi olla tietoturvan kannalta. Suurin ohjelointikielestä riippuva tekijä on kielen *muistiturvallisuus*, eli tunnistaako ajonaikainen ympäristö kiellettyjen muistivitteiden tekemisen. Nykypäivänä yleisimmät muistiturvallisuuden puutteesta kärsivät ohjelointikielet ovat C ja C++, joita tehokkuussyyistä edelleen käytetään laajasti [LE01].

Muistivirheet ilmenevät useimmiten prosessin tappamisena tietyllä käytötietorjelmäkohtaisella virhekoodilla. Unix-pohjaissä järjestelmässä tämä virhe on tunnettu 'Segmentation fault'. Koska muistivirheistä aiheutuu useimmiten ohjelman kaatuminen, mahdollistaa muistivirheen olemassaolo palvelunestohyökkäyksen eli DoS, denial-of-service-hyökkäyksen. Lisäseurauksena prosessin pakotetusta kuolemasta on se, ettei prosessilla ole mahdollisuutta siivota jotakin säilyvää tilaa, esimerkiksi väliaikastiedostoja. Hallitsematon kuolema voi myös aiheuttaa tiedon korruptoitumista jos ohjelma kaatuu esimerkiksi kesken levykirjoitusta. Kriittisimmät muistivirheet voivat mahdollistaa hyökkääjän suorittaa mielivaltaista koodia ohjelmaprosessin käyttööikeuksilla (RCE, Remote Code Execution).

Muistivirheiksi luokitellaan tyypillisesti seuraavat:

- Taulukon yli tai ali indeksointi: C-kielen taulukoissa ohjelmoijan vastuulla on huolehtia, ettei taulukkoa indeksoida liian suurella tai negatiivisella indeksillä. Jos näin pääsee tapahtumaan, kielen spesifikaatio ei ota kantaa siihen mitä tapahtuu. Käytännössä usein tapahtuu joko muistiviittaus taulukkoja ympäröiviiin muuttujaan tai ajonaikaympäristön tietorakenteisiin. Suurin riski pinossa olevan taulukon ohi kirjoitamisessa onkin aktivaatiotietueessa sijaitsevan funktion paluuosoitteeseen ylikirjoittaminen. Tästä seuraa käytännössä suoraan mielivaltaisen

koodin suoritus.

- Puskurin yliuoto: Vastaavasti kuin taulukoiden kanssa, C-kielessä täytyy merkkijonoja ja tavupuskureita kopioitaessa tai luettaessa ohjelmojan itse huolehtia, että puskurissa on riittävästi tilaa. Seuraukset ovat enimmäkseen samat kuin taulukon yli indeksioimisella. Puskureiden yliuoto on C:ssä harmillisen helppo tehdä. Jo C:n standardikirjastosta löytyy funktioita [LE01], joille ei ole ollenkaan mahdollista antaa tulospuskurin kokoa ja siten aiheuttavat yliuodon jos tulos ei mahdu puskuriin. Tällaisia funktioita ei koskaan pitäisi käyttää syötteen käsittelyn yhteydessä. Näistä syistä puskurin yliuodot pinossa ovat yliivoimaisesti yleisin syy RCE-haavoittuvuuksille.
- Alustamattoman muiston käyttö: C:ssä keosta tai paikallisille muuttujille varattua muistia alusteta mitenkään. Tällaisen muistialueen sisältönä on jotain mitä kyseisen muistialueen aiempi käyttäjä on sinne sattunut kirjoittamaan. Jos tällaista muistia sitten näytetään jossain muodossa käyttäjälle, saattaa käyttäjälle vuotaa joko ohjelman salaisia tietoja tai riittävästi tietoa ohjelman käyttämistä muistialueista helpottamaan RCE-haavoittuvuuden tekoa. ~~Ei~~ Yaihtoehtoisesti alustamaton osoitinmuuttuja johtaa usein ohjelman kaatumiseen.

Edellisten lisäksi C-ohjelmojan täytyy huolehtia dynaamisesti varatun muiston vapauttamisesta. Tästä aiheutuu vielä lisää mahdollisia virhetilanteita:

- Muistivuodot: Jos ohjelmoija unohtaa vapauttaa dynaamisesti varatun muistialueen sen jälkeen kun sitä ei enää käytetä, jää tuo muistivaraus kuluttamaan muistia ohjelman sulkemiseen asti. Tämä mahdollistaa esimerkiksi palvelunestohyökkäyksen, mikäli lisääntynyt muistinkäyttö johtaa ohjelman sivutukseen levylle.
- Muiston vapauttaminen useasti (nk. double free): Varattua muistialuetta ei saa vapauttaa kuin yhden kerran, koska esimerkiksi vapaut-

tamisen jälkeen sama muistialue voidaan antaa jonkin toisen muistivarausken käyttöön, mikä näinollen aiheuttaisi jonkin toisen, täysin liittymättömän muistialueen vapauttamisen. Yleensä vapautetun muistialueen vapauttaminen uudelleen aiheuttaa muistinhallintakirjaston sisäisten tietorakenteiden korruptoitumisen ja johtaa ulkopuolisen koodin suoritukseen [CGMN12].

3 Testausmenetelmät

Ohjelmistotekniikan menetelmistä tuttu laadunvarmistustekniikka on automaattiset testit [Som06]. Testausta voidaan soveltaa tietoturvaongelmien välttämiseen tietyn edellytyksin: sen sijaan, että testataan toivotun toiminnallisuuden olemassaoloa, testataan epätoivotun käytöksen puuttetta [PHP⁺¹¹]. Yleisempä epätoivottuja tapahtumia ovat kaatumiset ja jumiutumiset (esimerkiksi ikisilmukat).

3.1 Luokittelu

Testauskeinoja voidaan tavallisten funktionaalisten testien tapaan luokitella karkeasti *blackbox*- ja *whitebox*-testeiksi sen mukaan kuinka paljon testaaminen kohdistuu ohjelmiston tavanomaisiin rajapointoihin ja kuinka paljon ohjelmiston sisäisen rakenteen toimintaan [Som06]. *Farkkuna /kuva/sivu*
(laaja
lähde)

Blackbox-testauksessa ohjelmatiedostoa ajetaan aivan normaalisti, ja tutkitaan sen käyttäytymistä ohjelmaan sopivilla syötteillä. Esimerkiksi palvelinohjelmiston ollessa kyseessä siihen avataan verkkoyhteys, tai komentoriviohjelmalle annetaan syöte normaalisti komentoriviparametrien kautta.

Whitebox-testauskeinoissa kajotaan ohjelman sisäiseen rakenteeseen. Tämä minkälaiseen testaukseen on tapoja huomattavasti enemmän. Esimerkiksi jotkut keinot voivat vaatia pääsyä ohjelman lähdekoodiin, tai sitten ohjelman suoritusta voidaan analysoida tai muuttaa konekielitasolla.

3.2 Testauksen kriteereitä

Tarkastellaan seuraavaksi muutamia automaattista testausmenetelmää: *staattista analyysiä* ja *fuzzausta*.

4 Staattinen analyysi

Staattiseen analyysiin perustuvat keinot ovat lähdekoodin analyysiä tekeviä whitebox-menetelmiä. Tällaiset menetelmät yrityvät paikantaa lähdekoodista tyypillisiä ohjelointivirheitä. Monista tällaisista ongelmista saattaa olla seuraamuksia tietoturvan kannalta [BPCL09]. Yleisin tällainen keino on käänäjien tarjoamat varoitukset ohjelmiston käänösaikana. Usein käytettyjä staattisen analyysin työkaluja ohjelointivirheiden etsintään ovat *lint*-tyyliiset [Joh78] ohjelmistot eri ohjelointikielille sekä esimerkiksi Coverity [BBC⁺10].

4.1 Lint

Lint [Joh78] on varhainen C-kielisiä ohjelmia tarkistava staattisen analyysin työkalu vuodelta 1978. Lintin fokusena ei ollut tietoturvaongelmat, vaan yksinkertaiset ohjelointivirheet, C-kääntäjää tarkempi tyypintarkastus sekä siirrettävyysongelmat. Esimerkiksi:

- Alustamattoman muuttujan arvon käyttö.
- Käytämättömät muuttujat tai funktiot.
- Saavuttamattomat koodirivit.
- "järjettömät" vertailut. Esimerkiksi etumerkittömälle kokonaislukumuuttujalle x ei ehtolauseke `if (x < 0)` ole koskaan tosi.
- Yllättävä operaattoripredenssi. Esimerkiksi `if (x & 0x10 == 0)` näyttää päällisin puolin bittitarkistuksesta, mutta C:ssä ==-operaattori

~~Testauksen kriteereitä~~
~~Testauksen kriteereitä~~
~~Testauksen kriteereitä~~

sitoo &-operaattoria valvemmin, jollon ehtolausekkeen tulokseksi tulee aina 0.

vihainen koodas

Tuon ajan C-kääntäjissä käänämisen nopeus oli korkeammalla prioriteetilla, joen tämänkalaisetkin kriittiset tarkastukset ulkoistettiin Lint-ohjelmalle. Esimerkiksi C-kääntäjät tyypillisesti käsitlevät yhtä C-tiedostoa kerrallaan, kun taas Lint tarkastelee tiedostoja kokonaisuutena, mikä mahdolistaa tarkemman analyysin. Nykyiset käänäjät osaavat varoittaa edellämainituista virheistä jo käänösaikana.

Monet vastaanvalaiset staattisen analyysin työkalut eri kielille ovat nimetyt Lint-ohjelman mukaan, esimerkiksi JSLint JavaScript-kielelle. Erityisesti C-kielen tietoturvaongelmiin keskittynyt staattisen analyysin ohjelmisto on Splint (Secure Programming Lint), joka on aiemmin kulkenut nimellä LCLint.

Splint on keskittynyt löytämään samankaltaisia virheitä kuin Lint, mutta lisäksi myös muistivirheitä sekä muistivuotoja C-kielisistä ohjelmista. Muistivirheiden löytäminen tapahtuu esittämällä ohjelmaa constraint solving-ongelmana (mitä tämä on suomeksi?). Ohjelmaan on määritelty joista C-kirjaston puskurinkäsittelyfunktiota kohti joukko alkuehtoja, joiden täytyy päteä funktioita kutsuttaessa. Esimerkiksi kopioitaessa merkkijonoa `strcpy`-funktiolla ei osoitin kohde- tai lähdepuskuriin saa olla `NULL` sekä kohdepuskurin täytyy olla vähintään yhtä suuri kuin lähdepuskurin. Muistia varaa funktiot ja konstruktiot sen sijaan tuottavat jälkiehtoja - esimerkiksi onnistuneen `malloc`-funktiokutsun tuloksena on puskuri, jonka koko annettiin parametrinä. Mut jos Splint ei saa osoitettua, että alkuehdot pätevät aina, varoittaa se mahdollisesta puskurin ylivuodosta.

5 Fuzzaus

anglilainen

Fuzzaus on raa'an voiman keinona automaattiseen tietoturvatestaukseen. Fuzzauksen tarkoitus on generoida satunnaisia syötteitä testattavalle ohjelmalle siinä

u kokee (mitä)
toteutuu
ongelmat

Se perustuu mahdollisten syöteiden vaikan läpikäytön
tms. liertoiluun

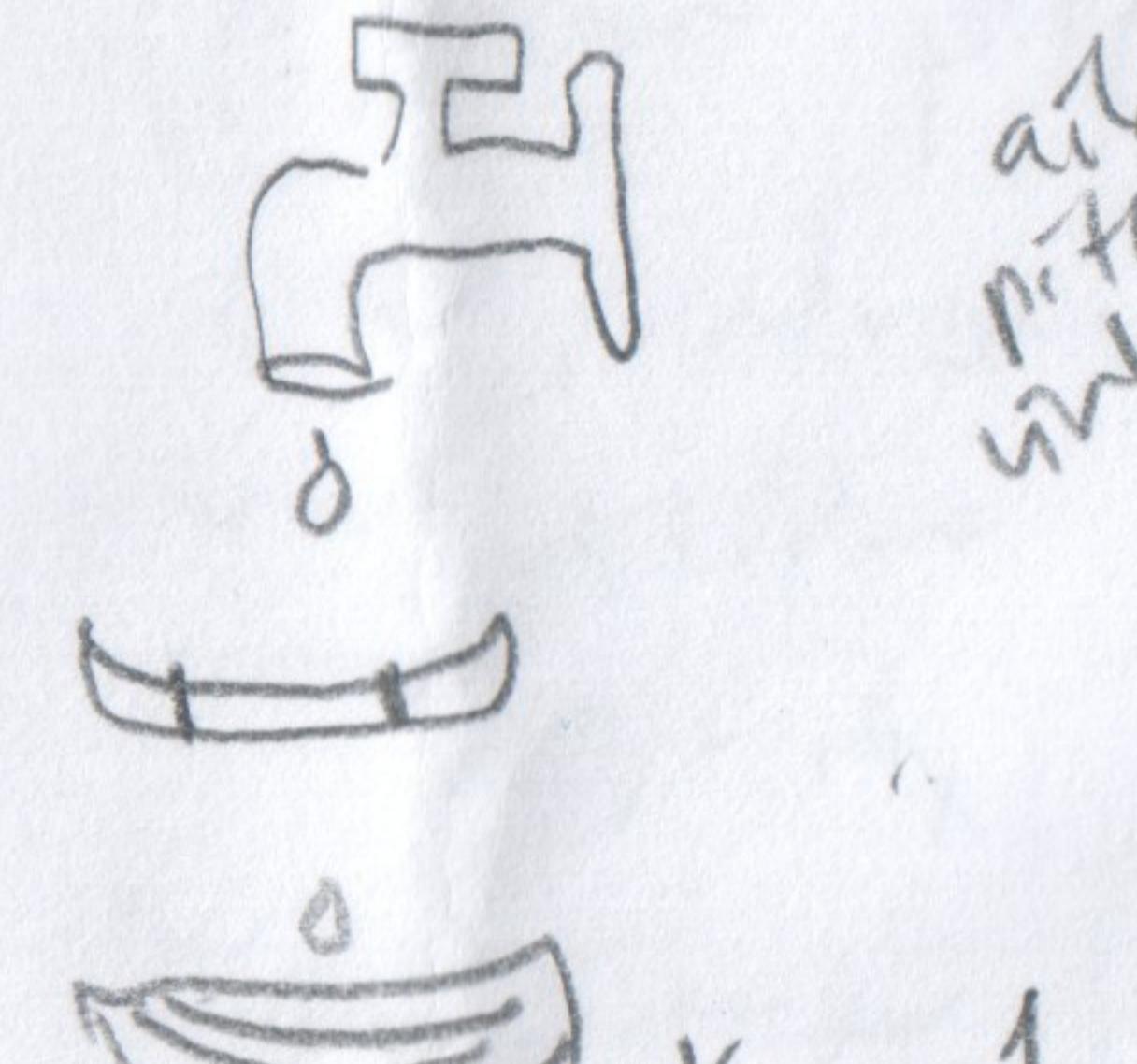
toivossa, että ohjelma ei käsittele niitä oikein [MFS90]. Muun muassa aiemmin mainittuja muistivirheitä löytyy usein fuzzauksella: normaalia reilusti pidemmät merkkijonot voivat yliuotaa kiinteäkokoisia puskureita. Samaten erikoisia merkkejä sisältävät merkkijonot sekä erittäin suuret tai negatiiviset lukuarvot voivat olla ongelmallisia [Oeh05] esimerkiksi yliuotamalla laskutoimituksia.

Fuzzaukselle hyvin sopivia kohteita ovat muun muassa erinäisten tiedostoformaattien jäsentäjät [GLM12, PHP⁺¹¹]: esimerkiksi web-selaimet joutuvat käsittelemään muun muassa HTML-, CSS-, PNG- ja JavaScript-muotoisia tiedostoja [PHP⁺¹¹]. Tämä on varsin suuri määrä koodia, joka ottaa syötteekseen tiedostoja suoraan lähtökohtaisesti epäturvallisesta Internetistä, jolloin hyökkäyspinta-alaakin on runsaasti.

Fuzzaukseen sopivien syötteiden luontiin on olemassa runsaasti tekniikoita aina yksinkertaisista blackbox-menetelmistä monimutkaisempin keinoihin:

- Yksinkertaisimillaan syöte voi olla lähes pelkkää satunnaista dataa. Esimerkiksi monien Unixin komentorivityökalujen syöte koostuu pelkistä tekstiriveistä ilman sen kummempaa rakennetta, jolloin jo no rivinvaihdolla eroteltua satunnaisgeneroituja tavuja [MFS90] on riittävä syöte ohjelmalle.
- Ennalta olemassa olevia kelvollisia syötteitä voidaan *mutatoida* lisäämällä, poistamalla, muokkaamalla jne. satunnaisesti.
- Rakenteellisesti (enimmäkseen) kelvollisia, mutta normaalista harvoin esiintyviä syötteitä voidaan luoda esimerkiksi kontekstittoman kieliopin tai jonkin muun formaalin syötteen määrittelyn perusteella.
- Ohjelman suoritusta voidaan analysoida symbolisesti välttämään syötteitä, jolla ei ole vaikutusta ohjelmaan [GLM12].

tämä
võib kriittise
luopeta kohdista
(enemmän saavutti
enemmän aikaa
tulenteikin)



Kuva 1.

8 kintekokoisten
puskureiden
ruotamisen

(tässä on koulukuntaroppi.)

(Johannes unielikuntakattomuudesta.)

5.1 Testitapausten generointi
6 Yhteenveto
Tässä tutkielmassa tarkasteltiin kahta menetelmää ohjelmistojen tietoturva-testaamiseen, staattista analyysia ja fuzzausta. Huomattavaa on, että koska osa tietoturvaongelmista on seurausta ohjelointivirheistä, niin tietoturvan parantamiseen soveltuu ohjelointivirheitä yleisesti paikantavan ohjelmistotekniikan menetelmät. Kuitenkin fuzzaus testausmenetelmänä on toimiva menetelmä erityisesti tietoturvaongelmien löytämiseen.

kohde?
kyröitä perusteella
muttaan, se on
osa kotimiestä
vestä
(mitä se f. tekee?)

Lähde: Su
oikene liitt
yhtenäisen
rikkeen

Lähteet

[BBC⁺10] Bessey, Al, Block, Ken, Chelf, Ben, Chou, Andy, Fulton, Bryan, Hallem, Seth, Henri-Gros, Charles, Kamsky, Asya, McPeak, Scott ja Engler, Dawson: *A few billion lines of code later: using static analysis to find bugs in the real world.* Commun. ACM, 53(2):66–75, helmikuu 2010, ISSN 0001-0782. <http://doi.acm.org/10.1145/1646353.1646374>.

[BPCL09] Baca, D., Petersen, K., Carlsson, B. ja Lundberg, L.: *Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?* Teoksessa *International Conference on Availability, Reliability and Security, 2009. ARES '09.*, sivut 804–810, maaliskuu 2009.

[CGMN12] Caballero, Juan, Grieco, Gustavo, Marron, Mark ja Nappa, Antonio: *Undangle: early detection of dangling pointers in user-after-free and double-free vulnerabilities.* Teoksessa *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*, sivut 133–143, New York, NY, USA, 2012. ACM, ISBN 978-1-4503-1454-1. <http://doi.acm.org/10.1145/2338965.2336769>.

[GLM12] Godefroid, Patrice, Levin, Michael Y. ja Molnar, David: *SA-GE: Whitebox Fuzzing for Security Testing.* Queue, 10(1):20:20–20:27, tammikuu 2012, ISSN 1542-7730. <http://doi.acm.org/10.1145/2090147.2094081>.

[Joh78] Johnson, S. C.: *Lint, a C Program Checker.* Teoksessa *Computer Science Technical Report*, nide 2B. Bell Telephone Laboratories, 1978.

[LE01] Larochelle, David ja Evans, David: *Statically detecting likely buffer overflow vulnerabilities.* Teoksessa *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10, SSYM'01*, sivut 14–14, Berkeley, CA, USA, 2001. USENIX Association. <http://dl.acm.org/citation.cfm?id=1251327.1251341>.

[MFS90] Miller, Barton P., Fredriksen, Louis ja So, Bryan: *An empirical study of the reliability of UNIX utilities.* Commun. ACM, 33(12):32–44, joulukuu 1990, ISSN 0001-0782. <http://doi.acm.org/10.1145/96267.96279>.

[MSC02] Moore, David, Shannon, Colleen ja Claffy, Kimberly: *Code-Red: a case study on the spread and victims of an Internet worm.* Teoksessa *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement, IMW '02*, sivut 273–284, New York, NY, USA, 2002. ACM, ISBN 1-58113-603-X. <http://doi.acm.org/10.1145/637201.637244>.

[Oeh05] Oehlert, Peter: *Violating Assumptions with Fuzzing.* IEEE Security and Privacy, 3(2):58–62, maaliskuu 2005, ISSN 1540-7993. <http://dx.doi.org/10.1109/MSP.2005.55>.

[PHP⁺11] Pietikäinen, Pekka, Helin, Aki, Puuperä, Rauli, Kettunen, Atte, Luomala, Jarmo ja Röning, Juha: *Security Testing of Web Browsers.* Communications of the Cloud Software, 1(1), joulukuu 2011. <http://www.cloudsw.org/under-review/ec2266cf-8d22-4bfe-a70c-3fa1569c7007>.

[Som06] Sommerville, Ian: *Software Engineering: (8th Edition) (International Computer Science).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006, ISBN 0321313798.

*) Ja kirjojen/konfliktien nimet menee isolle kun ne palauttaa läheteen

vihjeitä
otkissa evan
sinä olette kyllä
olle min piennellä
(paikallinen hyvä
vasta tätä)
Amerikkalaiset
tun. digyan
tunnus kirjaimia
(lähteiden hyvin
ei tuni
avaintestejä.
koodit har
informaatioita.
ei libnug)

/embed

encontra eten
tunne urli
ole muuttunut
finaliversioon