hyväksymispäivä arvosana

arvostelija

Ohjelmointivirheiden automaattinen etsintä

Taneli Rantala

Helsinki 21. huhtikuuta 2003

Tietoturvallisuus nykyaikaisessa liiketoimintaympäristössä - seminaarin esitelmä

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Ohjelmointivirheiden automaattinen etsintä

Taneli Rantala

Tietoturvallisuus nykyaikaisessa liiketoimintaympäristössä - seminaarin esitelmä
Tietojenkäsittelytieteen laitos
Helsingin yliopisto
21. huhtikuuta 2003,

Tässä esitelmässä esitellään kaksi menetelmä tietoturvaan liittyvien ohjelmointivirheiden automaattisen etsintään. Menetelmät ovat haavoittuvuusanalyysi ja ominaisuustestaus.

Aiheluokat(Computing Reviews 1991):

Avainsanat:

Sisältö

1 Johdanto

Tietoturvallisuuden varmistamiseksi eivät riitä pelkästään tehokkaat kryptografia-algoritmit ja suurimmat uhat muodostuvatkin ohjelmistojen heikkouksista: saatavien syötteiden kelvollisuutta ei tarkasteta, epäluotettavien kirjastofunktioiden käyttö ja niin edelleen. Haavoittuvuuksiin voidaan suhtautua kolmella tavalla:

- Haavoittuvuuksien välttäminen: käytetään ohjelmistonkehitystekniikoita, joilla on mahdotonta tuottaa haavoittuvia komponentteja.
- Haavoittuvuuksien eliminointi: etsitään haavoittuvuuksia ohjelmistosta esimerkiksi testauksella ja poistetaan ne.
- Haavoittuvuuksien sietäminen: kehitetään ohjelmiston sietokykyä haavoittuvuuksien suhteen, taataan oikea toiminnallisuus uhista huolimatta.

Ensimmäisestä vaihtoehdosta esimerkkinä voidaan pitää Javan käyttöä toteutuskielenä C/C++:n sijasta. Kahden viimeksimainitun kielen ongelmana ovat juuri puskurien ylivuodot ja virheelliset muistiviittaukset. Javassa tällaisia ongelmia ei ole, vaan virtuaalikone huolehtii kaiken muistin käsittelyn. Ongelmana ovat kuitenkin itse virtuaalikoneen mahdolliset haavoittuvuudet. Haavoittuvuuksien eliminoinnissa on ongelmana haavoittuvuuksien löytäminen: koodin monimutkaisuus ja koko ovat suhteessa sen sisältämään mahdolliseen virhemäärään. Haavoittuvuuksien sietämisestä esimerkkinä palomuurit (firewall), joka rajoittaa kommunikointia järjestelmän ulkopuolen kanssa ja tunkeilijoiden havaitseminen/torjunta (intrusion detection). Tämä esitelmä keskittyy tapaan kaksi. Edes virheetön suunnittelu ei pelasta toteutustason uhilta, joihin mm. edellä mainitut uhat kuuluvat. Toteutustason haavoittuvuuksia on kuitenkin helpompi havaita niiden yksiselitteisyyden takia kuin suunnittelutason. Työvälineitä haavoittuvuuksien automaattisen havaitsemiseen toteutustasolla on useita ja ne voidaan jaotella useiden ominaisuuksien ominaisuuksien mukaan: onko lähdekoodi tarpeellinen, onko testaus funktionaalista, analysoidaanko rakennetta staattisesti vaiko dynaamisesti ja niin edelleen.

Haavoittuvuusanalyysissa on käytetty lähteeä Ghosh et al. artikkelia [GOM98], ominaisuuspohjaisessa testauksessa Finkin ja Bishopin [?]. Johdannossa on käytetty kumpaakin sekä Kaksosen teosta [Kak01].

2 Haavoittuvuusanalyysi

Haavoittuvuusanalyysi pyrkii yhdistämään ominaisuustestauksen white-box paradigman ja suoritusaikaisen analyysin. Mahdolliset ongelmakohdat menetelmä etsii automaattisesti ja ottaa nämä erikseen testaviksi satunnaisilla merkkijonoilla. Merkkijonot lasketaan sekä normaaleista syötejakaumista että näiden käänteisjakaumista.

Esitettävä analyysi perustuu Ghoshin ja kumppaneiden AVA-algoritmiin, joka on implementoitu ainakin FIST-ohjelmistoon.

Alla: P on ohjelma, x on syötearvo, Q on normaali syötejakauma, R käänteinen syötejakauma ja S erikoissyötejoukko.

1. Jokaiselle sijainnille l ohjelmalle P suorita askeleet 2-7
2. Aseta counter = 0
3. Satunnaisesti valitse syöte x tai syötejono joukoista Q, R ja S. Jos P pysähtyy x:llä kiinteän mittaisen ajan aikana, etsi välittömästi l:n jälkeisten tilojen joukko. MerkitĎĎn joukkoa Z:lla.
4. Muunna valittua joukon Z muuttujan a:n arvoa ja merkitään joukkoa X:llä. Tämän jälkeen suorita jäljellä oleva koodi X:llä. Tapa, jolla a:ta muunnetaan on uhkajoukon T mukainen.
5. Jos P:n tulostiedot täyttävät predikaatin inkrementoi counter:ia
6. Toista askeleet 3-5 n kertaa missä n on testiatapausten määrä
7. Jaa counter n:llä, jolloin saadaan haavoittuvuusarvio l:lle. Arvoa 1 - haavoittuvuusarvio kutsutaan turvallisuusarvioksi.

Ongelmana algroitmissa sijaintien l, joihin syöttäminen tehdään, löytäminen. Käyttäjä voi tehdä tämän itse koodia selaamalla tai käyttämällä jotain sopivaa algoritmia. Vaiheessa 3 valitaan käytettävät testisyötteet. Syötteiden valinnassa täytyy ottaa huomioon kuinka todennäköisesti ne mahdollistavat onnistuneen tunkeutumisen: harvinaset tapaukset, jotka ovat tunnettuja, mutta vaarallisia syötteitä. Vaihessa 4 suoritetaan virheensyöttö eli ohjelman tilan "rikkominen"tai koodin muuntaminen. Vaiheessa 5 tarkastetaan aiheuttiko vaiheen 4 toimenpide tulosteen, joka on ristiriidassa predikaatin, eli turvaominaisuuden, kanssa. Tällöin turvarikkomusten määrää ($counter$) lisätään yhdellä. Turvaominaisuus ilmaistaan vakuutuksena (eng. assert), että ohjelma ei ole tietyssä tilassa.

Vaiheessa 7 tulostettu tieto järjestelmän turvallisuudesta ei ole absoluuttinen metriikka, kuten keskimääräinen vikaantumisaika (mean-time-to-failure) vaan suhteellinen. Käyttäjä voi verrata järjestelmän eri versioita tai eri järjestelmiä toisiinsa asettaen ne "paremmuusjärjestykseen".

Virheensyöttö ja FIST (Fault Injection Security Tool)

Edellä esitetty AVA-algoritmi on toteutettuna FIST-ohjelmistoon, joka suorittaa analyysin C ja C++ -kielillä kirjoitetuille ohjelmille. Tarkastelemme tässä luvussa virheensyöttökomponentin toimintaa (fault injection engine).

2.1 Virheensyöttöanalyysi

Virheensyötön sijoitus on prosessi, jossa lähdekoodia muunnetaan kelvottomien (eng. corrupted) tilojen ja poikkeksellisen käytöksen aikaansaamiseksi ohjelman suoritusaikana. Aluksi virheensyöttöfunktiot (eng. fault injection functions) asetetaan jokaiseen mahdolliseen sijaintikohtaan (AVA-algoritmin l). Testiajojen jälkeen saadaan selville tietoturvan kannalta kriittiset sijainnit. Tämän lisäksi yksittäisten muuttujien,lauseiden ja funktioiden suhteellisesta tärkeydestä sovelluksen tulosten kannalta. Esimerkiksi tulostusvirran käsittelyfunktioiden muuttaminen ei ole niin merkittävää kuin niiden funktioiden jotka käsittelevät käyttäjän syötetietoja.

FIST-ohjelmistossa on useita virhesyöttöfunktioita: primitiivisten tietotyyppien muuntamisesta, merkkijonojen manipulaatiot, puskurin ylivuotojen aikaansaamiseen ja edellämainittujen yhdistelmät.

Tietotyyppien muuntamisesta käy esimerkkinä Boolen vaihdokset (Boolean flips), joissa Boolen muuttujalle tehdään negaatio ja merkkien (eng. character) vaihdokset, joissa merkki korvataan satunnaisella ASCII:n mukaisella merkillä.

Merkkijonojen manipuloinnissa merkkijonoja katkaistaan (eng. truncate), katenoidaan satunnaisiin tai ennalta määrättyhin merkkijonoihin, luomalla täysin uusia merkkijonoja tai korvata ennalta annetuilla merkkijonoilla.

Puskurinylivuotofunktiot kirjoittavat sen pinokehyksen (eng. stack frame), johon puskuri on allokoitu, paluuosoitteen päälle itse puskurin osoitteen. Kehysosoitinta (eng. frame pointer) jäljittämällä virhesyöttöfunktio päättelee missä kohtaa kirjoittaa paluuosoitteen päälle. Ennen pitkää aktivointitietue, jossa muunnettu paluuosoite sijaitsee, otetaan pinosta pois ja ohjelma/prosessi siirtyy suorittamaan virheensyöttöfunktion määrittelemiä konekäskyjä. Käskyjä suoritetaan kuin ne olisivat osa prosessin/ohjelman normaalia toimintaa. Ongelmana funktioiden toteutuksessa ovat eri alustat: niiden tapa määritellä pinot vaihtelee. Turvattomat lukufunktiot, kuten C-kielen gets, strcat ja strcpy, mahdollistavat ylivuotohyökkäykset koska ne eivät tarkista kohdepuskurin kokoa. Yksi mahdollisuus on etsiä näitä tunnetusti turvattomia funktioita. Tämä ei kuitenkaan yksinään riitä, sillä ohjelmoijat käyttävät usein itsetehtyjä ja yhtä turvattomia lukufunktioita. Tämän takia FIST yrittää ylivuotoa myös ennalta tuntemattomissa lukufunktioissa. Lisäksi FIST-ohjelmalla voi yrittää ylivuotoa myös niissä muuttujissa, joita ei työnnetä pinoon. Vaikka tällöin ei tuloksena olekaan tietyn ohjelmakoodin suoritus, mahdollisuus esimerkiksi pääsyoikeuksien määrittelyyn käytettyjen muuttujien turmelemiseen on olemassa.

2.2 Esimerkki: NCSA httpd 1.5.2

Tässä luvussa tarkastellaan FIST:n käyttöä http-demoni-ohjelmiston analysoimiseen. Demoniohjelmat ovat yleisestikin mielenkiintoisia tietoturvakysymysten kannalta, koska ne tarjoavat palveluita epäluotetuille (eng. untrusted) käyttäjille. Demoniohjelmat toimivat usein super-user- tai root -tasoisilla oikeuksilla, jotta ne voisivat sitoa pistokkeita varattuihin portteihin ja/tai käydä läpi tiedostosysteemiä ilman rajoituksia. Tämän takia demoniohjelmat on oltava erityisen luotettavia, jotta ne eivät antaisi epäluotettaville käyttäjille korkean tason oikeuksia.

Tarkasteltava verkkodemoni on tarkoitettu hyperteksti-dokumenttien välittämiseen. Dokumentit voivat olla salasanalla suojattuja tai suojaamattomia. Tämä demoni oli valittu [], koska se oli havaittu haavoittuvaksi hyökkäyksille jo aiemmin [**?**, **?**]. Testiajoissa koetettiin hakea salasanalla suojattuja HTML-sivuja oikealla käyttäjätunnuksella, mutta väärällä salasanalla. Demonin kuusi komponenttia havaittiin haavoittuviksi virheensyöttöfunktiolle: http_access.c, http_auth.c, http_log.c, http_request.c, http_config.c ja util.c .

Modulin http_access.c funktio evaluate_access havaittiin haavoittuvaksi ongelmanaan mahdollinen suojatun sivun luvaton nouto:

329 num_dirs = count_dirs(path);

Tämä sijoitusfunktio kutsuu funktiota count_dirs , joka laskee hakemistojen lukumäärän dokumentin hakemistopolulla. Funktiota tarvitaan sen selvittämiseen onko asiakasohjelmiston pyytämä tiedosto on jossain pääsyvalvotussa hakemiston juuressa. Jos esim. pyydetty tiedosto on n:n ($n > 0$) syvyisessä hakemistossa dokumentin juuren suhteen, niin pyydettyä tiedostoa ei pitäisi antaa ilman asiakkaan autentikointia. Muuttujaa käytetään päätökseen, siitä kuinka monta hakemistoa läpikäydään käyttöoikeuksien tarkistusta varten. Jos muuttuja asetetaan pienemmäksi kuin sen hakemiston syvyys, missä pääsynvalvonta otettiiin käyttöön. Ongelma on siis sekä funktion count_dirs virheellinen toteutus että muuttujan num_dirs virheellinen luku / sijoitus.

Demonin aikaisemmassa versiossa (1.4.2) oli myös toinen haavoittuvuus, joka oli enemmän palvelunestotyyppinen (denial-of-service): saadessaan erikoismerkkejä yksi syöttöfunktioista lisäsi merkkijonoon koodinvaihtomerkkejä tarkistamatta muistin riittävyyttä. Näin asiakasohjelmisto pystyi kaatamaan palvelijan. Versiosta 1.5.2 tätä puutetta ei FIST:llä löydetty.

3 Ominaisuuspohjainen testaus

Ominaisuuspohjainen testaus (property-based testing) on Davisin yliopistossa kehitetty metodologia, jossa pyritään *validoimaan* toteutuksen virheettömyys tietyn *ominaisuuden* (property) suhteen. Yksittäiset ominaisuudet saadaan johtamalla ne spesifikaatioista, jotka voivat olla ilmaistuja formaalilla spesifiointikielellä kuten Z. Davisin yliopiston työryhmä on myös kehittänyt tähän menetelmään perustuvan työkalun *Testers Assistant*.

Ominaisuuspohjaisella testauksella on kahdeksan vaihetta:

1. Ominaisuuden valinta
2. Staattinen analysointi ja viipalointi (eng. slicing)
3. Ohjelman instrumentointi (eng. program instrumentation)
4. Alkutestitapausten luonti (eng. initial test cases) ja ajo
5. Peittyvyyden arviointi (eng. coverage evaluation)
6. Ylimääräisten testitapausten (additioinal test cases) valinta ja suoritus
7. Oikeellisuuden arviointi (eng. correctness evaluation)
8. Kolmen viimeisen askeleen toisto (lukumäärä riippuu ed. vaiheesta)

Tarkastelemme seuraavaksi jokaista vaihetta yksitellen mikä jälkeen katsomme menetelmän soveltamista SunOS 3.2:n ftp-demoniin.

3.1 Ominaisuuden valinta

Ominaisuus valitaan ensiksi geneeristen ominaisuusten kirjastosta (property library), minkä jälkeen se kirjoitetaan/muokataan ohjelmakohtaiseksi alemman tason ominaisuudeksi. Kummassakin ominaisuustyypissä koodauksessa TASPEC (Testers Assistant SPECification language). TASPEC on nimensä mukaisesti kehitetty tätä työkalua varten tavoitteenaan tarjota primitiivikonstruktiot viipalointia ja mahdollisten testioraakkelien luomiseen

Esimerkkinä geneerisestä korkeamman tason ominaisuudesta olkoon seuraava : *authenticated(uid)* **before** *permissions_granted(uid)* , missä *uid* on käyttäjätunnus.

Kirjastosta löytyvät kyllä alemman tason määritelmät kyseiselle ominaisuudelle:

location func *setuid(uid)* **result** 1 {
assert *permissions_granted(uid);* }

location func *crypt(password, salt)* **result**
encryptpwd { **assert** *password_entered(encryptpwd);*
}

location func *getpwnam(name)* **result** {
assert *user_password(name, pwent* $\rightarrow$ *pw_passwd, pwent* $\rightarrow$ *pw_uid); }*

location func *strcmp(s1,s2)* **result** 0 {
assert *equal(s1,s2)* }

password_entered(pwd1) $\wedge$
user_password(name, pwd2, uid) $\wedge$
equal(pwd1,pwd2) { **assert** *authenticated(uid);* }

Näissä määrittelyissä aaltosulkeiden sisällä olevat toimintojen täytyy ovat tosia vasta kun aaltosulkeita ennen oleva toiminto palauttaa **result** :n antaman arvon. Esimerkiksi **setuid(uid)** aiheuttaa *permissions_granted* predikaatin voimaantulon.

Autentikointi voidaan valita siis sekä manuaalisesti että automaattisesti. Automaattisessa versiossa toteutuksen lähdekoodia verrataan yo. koodipohjaan (code template). Predikaatin *permissions_granted* voimassaolo koodipohjassa vaatii **setuid**-systeemikutsua. Toteutuksen lähdekoodissa on siis oltava **setuid**, jotta predikaatin voimassaolo voitaisiin testata.

3.2 Staattinen analyysi ja viipalointi

Tässä vaiheessa toteutuksesta laaditaan tietovuokaavio, jolla on kolme on käyttötarkoitusta: ohjelman instrumentointi, ylimääräisten testitapausten generointi ja oikeellisuuden arvionti. Sitten ohjelma viipaloidaan halutun ominaisuuden mukaan. Tämä vähentää kaavion kokoa huomattavasti: esimerkiksi ftp-demoni sisältää viipaloimattomana 6148 solmua ja 31912 kaarta, autentikointiominaisuuden mukaan viipaloitu kaavio sisältää kuitenkin vain 9 tilaa ja 14 kaarta.

3.3 Ohjelman instrumentointi

Tässä vaiheessa Testers Assistant rakentaa vaihtoehtoisen version ohjelmasta ajettavaksi testiä varten. Vaihtoehtoisella versio toimii muuten samoin kuin alkuperäinen ohjelma, mutta siinä on ylimääräiset modulit testitulosten datankeruuta varten. Jokaisessa koodin osassa, joka vastaa yhtä ominaisuuden sijaintitunnistetta (eng. location specifier), on ylimääräinen ohjelmanpätkä tallennukseen siitä, jos ja kun osa suoritetaan. Lisättyä koodia käytetään myöhemmin oikeellisuuden arvioinnissa. Sijoituslauseet, jotka ovat tärkeitä peittyvyyden arvioinnissa merkitään, jotta tieto niiden suorituksesta tallennettaisiin. Viipalekohtaisen instrumentoinnin jälkeen ohjelma käännetään.

3.4 Alkutestitapausten ajo

Instrumentoitu ohjelma suoritetaan useaan kertaan eri testiaineistoilla. Testiaineistojen generoimiseen on kolme tapaa: testiaineisto, jota käytettiin tavanomaisessa testauksessa ja debuggauksessa, ohjelman funktionaalisesta kuvauksesta generoitu aineisto ja mahdollisista spesikaatioista luotu aineisto. Menetelmistä ensimmäinen on yksinkertaisin koska lisätyötä ei tarvita, tällainen testiaineisto onkin yleensä varsin kattava. Esimerkin ftp-demonin aineiston generointia varten voi käyttää esimerkiksi UNIX:n **man** -sivuja (funktionaalinen kuvaus). Demonille saadaan muun muassa seuraavat neljä tapausta:

```
user <user name>
pass <incorrect password>
retr filename

user <user name>
pass <correct password>
retr filename (no access permission)

user <user name>
pass <correct password>
cwd directory
retr filename1 filename2

user <user name>
pass <correct password>
list
```

, missä `user` tarkoittaa käyttäjätunnuksen syöttöä, `pass` salasanan syöttöä, `retr` tiedoston noutoa, `cwd` hakemiston vaihtoa ja `list` hakemiston listausta.

Testiajoille tehdään peittyvyys- ja oikeellisuusarvioinnit. Edellä mainitut ajot eivät riko autenktikointiominaisuutta. Peittyvyysarviointi paljastaa kuitenkin lisätestiaineiston luonnin ja ajamisen tarpeellisuuden kuten seuraavaksi nähdään.

3.5 Peittyvyyden arviointi

Jokaisen testiaineiston kohdalla Testers Assistant laatii viipaleen ajohistorian (eng. execution history), joka kertoo mikä ohjelman polku valittiin. Ajohistorioita verrataan peittyvyysmetriikkoja (eng. coverage metrics) vasten. Ominaisuuspohjaisessa testauksessa metriikkana käytetään iteratiivisia konteksteja. Konteksti on järjestetty jono sijoituksia, jotka määrittelevät ohjelman alipolun (eng. sub-path). Jotta historia olisi yhteensopiva kontekstin kanssa, täytyy sijoitusten tapahtua järjestyksessä ilman väliintuloja tai häiriöitä. Kontekstit saadaan staattisessa analyysissa luodusta tietovuokaaviosta.

Seuraavat ohjelmarivit on ftp-demonin esimerkin lähdekoodista:

```
(1) logged_in = 0;
(2) while(1)
(3) switch(cmd) {  (4) user: name = read();
(5) pass = read();
(6) if(match(name,pass))
(7) logged_in = 1;
(8) break;
(9) get: if(logged_in)
(10) setuid(name);
(11) }
```

Siihen liittyvät kontekstit ovat:

{ {(4,5,6,10} , {4,5,6,4,10} , {4,10,4,5,6} }

Ajohistorioita verrataan konteksteihin ja ne kontekstit jotka eivät kuvaudu ajohistorioiksi ovat peittyvyysaukkoja (eng. coverage gap).

Edellä mainittujen neljän testitapauksen ajohistoriat ovat:

{{4,10,4,5,6} , {4,5,6,10} , {4,5,6,10}, {4,5,6} } .

Ajohistorioista toinen ja kolmas ovat identtisiä, koska niiden käyttäytyminen suhteessa ominaisuuden määrittelyyn on identtinen. Konteksti {4,5,6,4,10} on peittyvyysaukko ja vastaa puuttetta toteutuksessa kuten seuraavaksi nähdään.

3.6 Ylimääräisten testitapausten luonti ja ajo

Jotta esimerkkimme peittyvyysmetriikka täyttyisi, ovat ftp-demonin ylimääräiset ajot tarpeen. Peittyvyysaukkoihin käytettävän uuden testiaineiston (automaattinen) luominen on ongelma. Esimerkissämme testaaja voi kuitenkin keksiä uuden testiaineiston vertailemalle peittyvyysaukkoja vastaaviin osiin koodia. Peittyvyysaukoissa voidaan havaita vastaavuus syötelauseiden ja lauseiden numeroiden välillä (lauseet 4 ja 5). Peittyvyysaukko voidaan siis paikata seuraavanlaisella aineistolla:

```
user <user 1's name>
pass <user 1's password>
user <user2>
pass <random string>
retr filename
```

Tämän testiaineiston oikeellisuusarviointi paljastaa puutteen demonissa, kuten seuraavaksi nähdään.

3.7 Oikeellisuuden arviointi

Jokaisen testiajon kohdalla Testers Assistant tallentaa tiedostoon aktivoidut TASPEC-primitiivit. TASPEC:n evaluointiohjelmisto prosessoi tämän datan ja vertaa sitä ominaisuuden määrittelyihin. Käyttäjälle annetaan ilmoitus mikäli ominaisuutta rikotaan. Edellä mainitun ylimääräisen testiajon aikana Testers Assistant rekisteröi käyttäjän 1 oikeelliseen autentikoinnin , mutta ei käyttäjän 2, koska salasanan tarkistus epäonnistuu. Kun tiedoston nouto tapahtuu, rekisteröidään ominaisuus *permissions_granted*. Kuitenkin nouto tapahtuu käyttäjän 2 oikeuksilla. Tämä on kuitenkin rikkomus autentikointiominaisuutta kohtaan, koska käyttäjää 2 ei oltu autentikoitu.

Viitteet

GOM98 Ghosh A.K, O'Connor T., McGraw G. "An automated approach for Identifying Potential vulnerabilities in software". Proceedings of the IEEE Symposium on Security and Privacy, May 3-6, 1998. Oakland, CA USA.IEEE. pp.104-114

Kak01 Kaksonen, R.
"A Functional Method for Assessing Protocol Implementation Security"(Licentiate thesis). Espoo. Technical Research Centre of Finland, VTT Publications

FiB97 Fink G., Bishop M. "Property Based Testing: A New Approach to Testing for Assurance,"ACM SIGSOFT Software Engineering Notes, 22(4) (July 1997).