FOA
FÖRSVARETS
FORSKNINGSANSTALT

FOA-R--98-00807-503--SE
April, 1998
ISSN 1104-9154

Användarrapport

Mats Persson

Mobil kod och säker exekvering

Avdelningen för Ledningssystemteknik
Box 1165
581 11 LINKÖPING

Försvarets Forskningsanstalt
Avdelningen för Ledningssystemteknik
Box 1165
581 11 Linköping

FOA-R--98-00807-503--SE
April, 1998
ISSN 1104-9154

Mobil kod och säker exekvering

Mats Persson

Projektnummer: E7023
Sändlista: FMV:Infosyst, HKV, FHS, LSC, FOA: 73, 12, 62

Dokumentets utgivare
Försvarets Forskningsanstalt
Avdelningen för Ledningssystemteknik
Box 1165
S-581 11 Linköping

Dokumentnamn och dokumentbeteckning
FOA-R--98-00807-503--SE

Dokumentets datum
April, 1998

Projektnr
E7023

Projektnamn (ev. förkortat)
Försvarsspecifik informationssäkerhet

Upphovsman (-män)
Mats Persson

Uppdragsgivare
Försvarsmakten

Projektansvarig
Alf Bengtsson

Fackansvarig
Alf Bengtsson

Dokumentets titel
Mobil kod och säker exekvering

Huvudinnehåll

Denna rapport visar och studerar de säkerhetsproblem som uppstår om man låter främmande mobil programkod skickas över ett datornätverk för att köras på andras eller egna datorer. Det finns ett antal programspråk och teknologier där man försökt lösa dessa problem på olika sätt.

Den säkerhetsmodell som de flesta av dessa system använder är den så kallade sandlådemodellen. Den innebär att man kapslar in koden i en begränsad omgivning där den inte kan ställa till med så mycket skada. Koden kommer inte åt variabler och minne utanför sandlådan, den kan ha begränsningar på tid och minnesutrymme som den får exekvera i, och den får begränsad tillgång till filsystem och nätverk via ett mer eller mindre avancerat behörighetssystem.

De system som undersökts lite noggrannare är programspråken Java, SafeTcl och Perl, samt ActiveX teknologin och operativsystemet Unix. Alla dessa system har någon form av sandlåda. Ingen av dem uppfyller alla de krav som man kan ställa på en fullgod sandlåda.

Nyckelord
datasäkerhet, scripts, sandlådemodell, Java, ActiveX, SafeTcl

Övriga bibliografiska uppgifter

Språk
Svenska

ISSN
1104-9154

ISBN

Omfång
43 sidor

Pris
Enligt prislista

( ) Begränsad distribution

Distributör (om annan än ovan)

Issuing organization
Defence Research Establishment
Division of Command and Control
Warfare Technology
P.O. Box 1165

Document name and doc. ref. No.
FOA-R--98-00807-503--SE

Date of issue
April, 1998

Item designation
E7023

Project name (abbreviated if necessary)
Information Security

Author
Mats Persson

Initiator or sponsoring organization
Swedish Defence

Project manager
Alf Bengtsson

Scient. and techn. responsible
Alf Bengtsson

Document title
Mobile code and safe execution

Abstract

This report shows and studies the security problems that arise when you send unknown mobile program code over a computer network to be executed on your own or someone elses computer. There are a number of programming languages and technologies that tries to solve these problems in different ways.

The security model that most of these systems use is the so called sandbox model. In this model the code is encapsulated in a limited environment where it can't do much damage. The code can't access variables and memory outside the sandbox; it can have limits on time and memory where it can execute in; and it has a limited access to filesystems and networks using a more or less advanced access control system.

The systems that are examined in more detail are the programming languages Java, SafeTcl and Perl, and also the ActiveX technology and the operating system Unix. All these systems have some form of sandboxes. None of them fullfills all the requirements you can make on a adequate sandbox.

Key words
computer security, scripts, sandbox model, Java, ActiveX, SafeTcl

Further bibliographic information

Language
Swedish

ISSN
1104-9154

ISBN

Pages
43 pages

Price
According to price list

( ) Restricted distribution

Distributor (if not issuing organization)

Innehåll

1 Inledning

Internet var från början en militär uppfinning och idén var att det skulle bli ett robust och säkert nätverk av sammankopplade datorer. Denna uppfinning började även användas av den kommersiella och civila världen, och på senare år har användningen av internet och dess tjänster ökat kraftigt. World Wide Web och epost har blivit välkända begrepp även hos vanligt folk.

En uppgift för FOA är att utreda hur nya ledningsystem skall utformas och hur det "digitala slagfältet" kommer att se ut. Av flera skäl bör man om möjligt utnyttja de kunskaper och den teknik som finns ute på den civila sidan. Man vill alltså på något sätt använda det internet som finns idag och anpassa den till försvarsspecifika krav.

Man vill fortfarande ha ett robust och säkert system, och dessutom mobilt. Tyvärr är den teknik som används i internet just nu ganska osäker och inte särskilt mobil. Utvecklingen går i ett snabbt tempo och ekonomi och användarvänlighet kommer i första hand. Därför kan det vara lämpligt att försvaret och FOA fokuserar på säkerheten i nätverk och hos datorer. Den här rapporten tar upp ett av säkerhetsproblemen men innan dess en kort översikt och historik om säkerheten på internet.

De fyra stadierna

I början av internet på 1970-talet användes det för att skicka elektronisk post, föra över filer och logga in på andra datorer. Internet användes mest av forskare. Det varken fanns eller behövdes någon avancerad säkerhet eftersom man litade på varandra. Det gick lätt att förfalska epost eller avlyssna en filöverföring eller lösenordsinmatning, men detta sågs inte som något stort problem.

Det andra stadiet började omkring 1990 när World Wide Web introducerades. Detta gjorde internet åtkomligare för fler människor eftersom det blev lättare att använda. Folk kunde göra egna hemsidor och företag kunde göra reklam för eller sälja produkter. Mängden information som skickades över internet mångdubblades. Två nya säkerhetsproblem uppkom. För det första, att fler kände till internet och risken för missbruk ökade därmed. För det andra, problemet var att internet blev för homogent, det vill säga för mycket folk använde samma programvara. Ett allvarligt fel i till exempel webläsaren Netscape skulle kunna få katastrofala konsekvenser.

World Wide Web var nu ett sätt att organisera och distribuera information, men det saknade interaktivitet. Det var svårt för användaren att bestämma hur denne ville ha informationen presenterad och det var svårt att söka. Detta ledde till det tredje stadiet då specialprogram för att interagera med användare introducerades. De kallades CGI-

scripts och körde på informationsdatabaserna och till dessa kunde användaren skicka in en specialiserad förfrågan om information varefter CGI-scriptet skickade tillbaka en sida med den efterfrågade informationen. Nu uppkom ännu ett säkerhetsproblem. Det var ofta enkelt för programmerare att skriva och lägga in dessa program, vilket gjorde att programfel och påföljande säkerhetshål blev ännu vanligare. CGI-scripts hade ibland större behörigheter än programmerarna själva och blev därmed en vanlig ingångsport för otillåtna intrång.

Nästa naturliga steg var att skicka över programmen till användarnas datorer och låta det köras där. Det gav stora prestandavinster både vad gäller datorkraft och mängden överförd data. De två populäraste metoderna att skicka program är med hjälp av så kallade applets skrivna i Java eller att man skickar ActiveX objekt. Att låta främmande programkod köra på ens egen dator innebär förstås ytterligare säkerhetsproblem. Det är i detta fjärde stadium som internet befinner sig just nu och det är också det den här rapporten kommer att handla om.

Problemen

Tyvärr finns många av de gamla säkerhetsproblemen på internet kvar och nya tillkommer hela tiden. Det är fortfarande möjligt att förfalska epost och avlyssna lösenord. Det är lätt att överbelasta och sänka en dator genom att skicka en extremt stort antal anrop till den, och det finns fortfarande en stor mängd säkerhetshål där man kan få otillbörliga rättigheter. Till detta tillkommer problemet att utvecklingen inom området går mycket snabbt och datasäkerheten är det som oftast släpar efter.

Lösningarna

Den säkraste lösningen på alla dessa problem är förstås att koppla ur datorn från internet, låsa in den i ett valv och inte låta någon ha tillgång till den. Det är en dålig lösning eftersom man inte har någon nytta av datorn då. Det näst bästa är de lösningar som har dykt upp på senare år med krypterade inloggningar och förbindelser, identitetskontroll med hjälp av digitala certifikat och brandväggar för att stänga ute angripare. En ansats till lösning på problemet med att låta främmande kod köra på ens egen dator är den så kallade sandlådemodellen som kommer att tas upp i den här rapporten.

Läsanvisningar

Denna rapport är främst riktad till de som arbetar med försvarets ledningssystem eller är intresserade av säkerhetsproblemen på internet. För att kunna ha ett större utbyte av resten av rapporten förutsätts en del kunskaper om internet, datorer och programspråk. Det finns dock en del enklare översikter om en del ämnen.

Rapporten kommer att i kapitel 3 beskriva eller förklara olika begrepp såsom autenticering, integritet och mobila scripts. I kapitel 4 kommer sandlådemodellen att beskrivas mer i detalj och en del säkra programspråk som Java och SafeTcl kommer att beskrivas och jämföras. Appendix i kapitel 5 kommer att ta upp andra relaterade system på internet, till exempel Netscape och SSL.

2 Sammanfattning

Denna rapport visar och studerar de säkerhetsproblem som uppkommer om man låter främmande mobil programkod skickas över ett datornätverk för att köras på andras eller egna datorer. Det finns ett antal system där man försökt lösa dessa problem på olika sätt och jag beskriver dessa i den här rapporten. Jag har även försökt få en sammantagen helhetsyn på problemen och deras möjliga lösningar.

Eftersom försvarsmakten har valt att använda ett datornätverk liknande internet eller rentav utnyttja internet, så är det lämpligt att studera vilka säkerhetsproblem man kommer att råka ut för i så fall. Det andra valet är att inte koppla upp datorer i nätverk men då förlorar man förstås möjligheten att snabbt och effektivt skicka information. Det vore som att låta bli att använda radio och telefoner.

Att man kör program på sin dator som utomstående har skrivit har förstås alltid förekommit. Det är det normala och vanligaste förfarandet. Oftast använder man bara program från kända tillverkare eller kända personer som man litar på, och dessa skulle förlora kunder om det visade sig att deras program betedde sig elakt. Dessutom kör man ofta sina program i ett operativsystem som har särskilda skyddsmekanismer.

Den senaste utvecklingen på internet och World Wide Web har gått mot att skicka programkod över nätverket i form av scripts eller objekt. Dessa skickas ofta helt automatiskt fram och tillbaka utan att användaren vet varifrån de kommer eller vem som skrivit dem. Denna programkod får ofta samma rättigheter som andra program och därmed full tillgång till hela datorn. Säkerhetsriskerna med detta är uppenbara och kan leda till katastrofer.

Ett annat relaterat problem är de virus och macrovirus som sprids i PC och Mac-världen. Numera kan de sprida sig snabbt via internet när användare läser sin epost eller tar hem dokument och därmed kan deras dator blir smittad av datorvirus.

Det finns ett antal programspråk och teknologier där man försökt lösa säkerhetsproblemen vid exekvering av mobil kod. De har lite olika angreppssätt och har fokuserat på olika risker. Den säkerhetsmodell som de flesta av dessa system använder är den så kallade sandlådemodellen. Den innebär att man kapslar in koden i en begränsad omgivning där den inte kan ställa till med så mycket skada. Koden kommer inte åt variabler och minne utanför sandlådan, den kan ha begränsningar på tid och minnesutrymme som den får exekvera i, och den får begränsad tillgång till filsystem och nätverk via ett mer eller mindre avancerat behörighetssystem.

Till sandlådan kan man även lägga identitetskontroll av vem som skrivit koden och varifrån den kommer och sedan använda sig av detta för behörighetskontroll. I en del implementationer av sandlådemodellen görs integritetskontroll av kod och data, och kryptering av kommunikationskanalerna.

Efter att ha undersökt och studerat de olika systemen och deras sandlådor har jag formulerat och sammanfattat ett antal krav på en säker exekveringsomgivning och dessa finns i kapitel 4.2.1.

De system som undersökts lite noggrannare är programspråken Java, SafeTcl och Perl, samt ActiveX teknologin och operativsystemet Unix. Alla dessa system har någon form av sandlåda. Ingen av dem uppfyller samtliga krav som jag formulerat.

Java och SafeTcl är de två programspråk som bäst uppfyller kraven och har därmed en ganska säker sandlåda. Trots att Unix inte är designat efter sandlådemodellen och samtidigt är ett äldre och primitivare system så har det skyddsmekanismer såsom identitetskontroll och behörighetsnivåer. ActiveX har bara identitetskontroll och uppfyller inga andra krav och kan därmed betraktas som ganska osäkert. Perl, eller Penguin som det egentliga systemet kallas, uppfyller många av kraven men är än så länge bara på experimentstadiet.

Java, Javascript och ActiveX finns implementerat i flera olika webläsare. Om man har ett kritiskt och känsligt system och vill öka säkerheten bör man stänga av och undvika Java, Javascript och ActiveX. När man använder sin webläsare för privat hemmabruk där man inte lagrar känslig information så är de förmodligen tillräckligt säkra.

Det finns alltså fortfarande inte något lättanvänt, flexibelt och säkert system för exekvering av programkod. Det är kanske inte så förvånande eftersom internet och främst World Wide Web är ett väldigt nytt område. Det kommer att ta lång tid för den evolutionära processen att förändra arkitekturen och utveckla ett system där risken för intrång och förstörelse är så låg att man vågar använda det för militära system. För att komma dit måste man utveckla en bra säkerhetsmodell och en säkrare arkitektur. Utgående från studierna i den här rapporten hävdar jag att det är möjligt.

3 Allmänt

Detta kapitel innehåller en bakgrund till resten av rapporten. Det tar upp en del begrepp och definitioner av datasäkerhetstermer. Flera av dessa termer har olika definitioner och åsikterna går isär om hur de skall tolkas. Det finns inte ens en gemensam och allmän definition av begreppet datasäkerhet.

"A computer is secure if you can depend on it and its software to behave as you expect." - Practical Unix & Internet Security [11]

3.1 Historisk bakgrund

Innan jag börjar med de grundläggande säkerhetsbegreppen så tänkte jag göra en lite personlig betraktelse av de historiska problemen kring datasäkerhet, programspråk och exekvering av program.

Hur skulle programmiljön sett ut om säkerheten vore ett icke-problem? Om alla datorprogram vore hårdlödda in i datorn och alla data var enbart text eller siffror. Då skulle det knappast finnas några säkerhetsproblem. Förr i tiden på 50- och 60-talen var program kompilerade en gång för alla och programmerare och systemchefer visste var programmen kom ifrån och hur de fungerade. Samtidigt som de hade kunskap om det statiska systemet så hade de kontroll över det. De vanliga användarna kunde inte göra mycket annat än mata in data och få ut ett resultat utan någon kontroll över processen. I stort sett var inte exekvering av program något direkt säkerhetsproblem.

En ganska naturlig utveckling var att användarna ville kontrollera exekveringen och det var då problemen började. Samtidigt insåg programmerarna fördelen med dynamiska program som kunde modifiera sig själva eller hämta och exekvera kod från andra källor och ha intern tolkning av kommandorader. Samtidigt som data kunde innehålla text så kunde det innehålla instruktioner för hur det skall tolkas av datorn.

Exekvering blev oförutsägbar inte bara på grund av att användarna själva kunde skriva kommandon utan även genom att hela systemet blev dynamiskt.

För att lösa problemet med att flera olika användare skulle kunna starta program när de behövde så konstruerade man operativsystem. Man byggde system för resurshantering och för att behålla säkerheten gav man användarna begränsad access till resurserna, program-

mens access begränsades, och åtkomstskydden sköttes av särskilda program. Man gick alltså från ett ganska säkert enanvändarsystem till ett mer användarvänligt men något osäkrare.

Nyare operativsystem har baserats på ett enkelt system med ett användarvänligt gränssnitt där en användare kan göra vad han vill med datorn och som nästan helt saknar säkerhet. Det har förstås lett till efterföljande problem med virus och liknande och som en efterkonstruktion har man lagt till viss säkerhet men den utgör inte grunden i operativsystemet.

I och med internets utveckling har vi fått möjlighet att hämta program med FTP (File Transfer Protocol), skicka post med email, debattera på News eller surfa omkring på World Wide Web. Från början var WWW ett enkelt system för att skicka sidor med text och sedan har man lagt till flera olika möjligheter att skicka exekverbar kod. Inte bara HTML (Hyper Text Markup Language) med dess begränsade kod, utan även fullständiga programspråk.

3.2 Grundläggande säkerhetsbegrepp

Det finns ett antal grundläggande säkerhetsbegrepp och de fem viktigaste beskrivs kortfattat här. Alla dessa begrepp hör ihop och sammantaget bildar de grunden för datasäkerhet. Deras specifika användning inom säker exekvering av mobila scripts tas också upp.

3.2.1 Autenticering

Begreppet autenticering har under senare tid delat upp sig i två betydelser. Den ursprungliga betydelsen är att visa att något är äkta och inte någon kopia eller ett falskt original.

På senare tid har begreppet även fått betydelsen identitetskontroll, vilket innebär att man bevisar att den identitet som uppges är äkta. Detta kan gå till på olika sätt. En person kan styrka sin identitet med till exempel id-kort där man kontrollerar att fotot stämmer överens med personen eller att man anger ett lösenord när man loggar in. Det kan även innebära att man bevisar att en särskild mängd data är äkta och är skriven av en viss person eller kommer från en viss plats. När det gäller mobila scripts så är autenticering med hjälp av digitala signaturer vanligast och för identifiering av personer används digitala certifikat.

I de digitala signaturerna använder man sig av asymmetriska nycklar och hashfunktioner för att bevisa identiteten. Hur detta fungerar kan man läsa mer om i Applied Cryptography [6].

Det viktiga när det gäller mobila scripts är att det finns många inblandade parter man vill autenticera. Man vill kunna visa vem som har skrivit scriptet och vem som skickat det och man vill även kunna identifiera från vilken dator det kommer ifrån, till vilken dator det skickas och i vilken omgivning scriptet kommer att köras. Med hjälp av denna autenticering kan man sedan ge scriptet behörigheter vid exekvering eller kontrollera att ingen ändrat i det. De digitala certifikaten och deras publika och privata nycklar används även för konfidentialitet när man skickar krypterade data över ett nätverk.

3.2.2 Auktorisering

Med auktorisering menas att ge behörighet eller tillåtelse att göra något och ibland kallas det för åtkomstskydd eller behörighetskontroll. Några exempel på operationer där man behöver kontrollera behörigheten är att läsa en fil, starta ett program eller öppna en nätverksförbindelse. Behörighetskontrollen kan ske vid inloggning och gäller därefter resten av inloggningstiden, eller för att få det säkrare kan man kontrollera varje gång en operation skall utföras. Denna kontroll börjar med en autenticering varefter man ur en lista eller matris hämtar de behörigheter denna person, dator eller program har. Några exempel på dessa listor och hur de fungerar finns i kapitel 5.7. Ett villkor för att detta skall fungera är att listorna också är skyddade för ändring i sig själv. När denna behörighetskontroll är gjord kan personen börja sin operation.

Andra sätt att tilldela behörigheter är att gruppera personer, filer eller program och sedan ge dessa grupper särskilda behörigheter. Man kan även ge en person en roll som då har sina egna behörigheter. Flera personer kan ha denna roll.

När det gäller mobila scripts så behöver dessa också behörigheter när de ska exekvera på andra datorer och det kan bli ett ganska komplext system eftersom det är många aktörer inblandade. Det bör också finnas regler för hur nya behörigheter skapas eller ärvs av nya scripts. Behörighetsystem kommer också att användas i sandlådemodellen i kapitel 4.2 där man då avgör hur de script som exekverar får tillgång till yttre resurser.

Det kan också vara lämpligt att ge behörigheter för nätverket och man anger då vilka personer eller program som får öppna och sända på kanaler.

3.2.3 Konfidentialitet

Konfidentialitet eller sekretess betyder att man håller något hemligt eller privat, vilket i det här sammanhanget ofta innebär kryptering. I ordets ursprung “konfident” finns en betydelse säker eller pålitlig, vilket betyder att man litar på kommunikationsvägen och att den är säker.

När det gäller programkod och scripts som skickas över nätverket så är kryptering inte lika viktigt eftersom passiv avlyssning av programkod inte kan ställa till med lika stor skada som aktiv ändring av koden. Därför är integritet viktigare.

Ett kommunikationsprotokoll som innehåller autenticering, kryptering och integritetskontroller är Secure Sockets Layer (SSL). Det beskrivs närmare i kapitel 5.2.

3.2.4 Integritet

Integritet betyder bevarande av värdet eller skydd av data från ändring. I detta ingår även att kunna upptäcka om något har ändrats i data.

Integriteten kontrolleras genom att data körs genom en envägs hash-funktion och sedan stämpla värdet från hash-funktionen med hjälp av den privata nyckeln hos avsändaren. Detta resulterar i en digital signatur som kan kontrolleras av mottagaren genom att köra data genom samma hash-funktion och sedan jämföra resultatet med den dekrypterade digitala signaturen. Är de lika så har datats integritet bevarats.

För programkod är det viktigt att den inte ändras och speciellt gäller detta maskinkod där en liten ändring kan få stora konsekvenser. Detta till skillnad från en vanlig textmassa där man måste göra ganska stora förändringar innan texten blir obegriplig för en människa.

Inte bara integriteten för programkod som skickas över nätverket behöver upprätthållas utan även den kod som ligger lagrad på disk. Speciellt gäller detta de säkerhetskritiska delarna.

3.2.5 Oavvislighet

Oavvislighet innebär att personen inte skall kunna förneka att denne har utfört en viss operation, som till exempel sänt eller tagit emot ett meddelande. Detta löser man ofta genom loggning av operationer eller tidsstämplar av meddelanden, som sedan registreras på en annan säker plats tillsammans med en signatur för personen.

När det gäller scripts är det bra att kunna logga deras exekvering, för att sedan kunna spåra och bevisa vilka operationer de utfört.

3.3 Exekverbart innehåll

När det gäller programkod och text som skickas över ett nätverk så är det viktigt att veta eller ännu hellre kontrollera vad som händer med det som skickas. Det är naivt att tro att allt som är menat som text alltid kommer att behandlas som text. Det som är exekverbart kommer med stor sannolikhet att exekveras någonstans och förmodligen är denna ibland godtyckliga exekvering av data en av de största källorna till säkerhetsproblem på internet. Det är därför nödvändigt att specificera och definiera begreppet.

Med begreppet exekverbart menas att texten eller data kan tolkas eller förstås av en dators processor eller av ett annat program. Det motsvaras av begreppet läsbart som man använder om sådan text som människor kan läsa och förstå. Men det finns en dualitet mellan kod och text. En rad av tecken kan alltså både förstås av människor och datorer vilket egentligen är ett krav på ett bra programspråk. En annan sak man bör tänka på är att all text som passerar en dator, undersöks eller processas på något sätt. Den kan till och med råka bli exekverad även om den bara var tänkt som text.

I den här rapporten används begreppet kod för data som är exekverbar och text för det som är avsett för människor.

Det finns egentligen olika grader av exekverbarhet och jag har delat upp dem i ett antal typer med stigande exekverbarhet. Egentligen är det en glidande skala och uppdelningen är mer gjord för att visa på skillnader i exekverbarhet.

- **Text** som innehåller information läsbart för människor. Eventuellt skulle ett avancerat AI-program kunna tolka den, men den innehåller ingen direkt strukturerad information utan bara ren text.
- **Märken** är en slags instruktioner för hur en text skall formatteras eller tolkas av till exempel en ordbehandlare. Det är ofta bara ett par tecken för att till exempel markera att en visst ord skall vara kursivt. Ett exempel på denna slags formateringspråk är HTML och ett annat är själva den här uppräkningen av exekverbarhet i Framemaker.
- **Makron** är en instruktion som genererar en text. Ett makro kan se ut som <datum> och när den läses av en dator så ersätts det med dagens datum. Andra makron kan även beräkna aritmetiska uttryck eller numrera sidor.
- **Styrkommandon** är instruktioner för hur data skall tolkas. Det kan ange villkorlig text, definition av variabler eller ändringar av dem. De kan styra beteendet av programmet som tolkar data.
- **Scripts** är skrivna i fullständiga interpreterande programspråk som är turingmaskinekvivalenta. En turingmaskin är ett datalogiskt begrepp för den enklast möjliga teoretiska representationen av en dator. Scripts har alltså möjlighet att styra exekveringspunkten med hjälp av loopar och hoppinstruktioner, och de kom-

mer åt systemfunktioner, filsystem och nätverk.
- **Bytekod** är instruktioner som tolkas av en virtuell processor, en slags datorsimulerad maskin, men i övrigt fungerar det som maskinkod.
- **Maskinkod** är instruktioner åt processorn i en dator. De är inte särskilt läsbara och egentligen det enda som en dator förstår. Maskinkod är ofta resultatet efter en kompilering.

Det finns två viktigare begrepp som kommit upp och det ena är interpretering vilket innebär tolkning av ett script och det andra är kompilering vilket innebär att man översätter från ett format till ett annat, programtext till maskinkod.

3.4 Mobila scripts

Ett mobilt script är ett program som körs någonstans i ett distribuerat nätverk eller datorsystem oftast med kontroll över var det körs och vart resultatet skall skickas. Den som startar scriptet behöver inte själv vara ägare till det utan kan starta det med fjärrstyrning. Det behöver heller inte vara en användare som startar scriptet utan detta kan ske automatiskt. Man kan klassificera mobila scripts i ett antal typer.

- **Lokalt script** startas av användaren själv och körs på den egna datorn och är egentligen inte att betrakta som mobilt. Detta är det normala sättet att starta program.
- **CGI** betyder Common Gateway Interface och har blivit ett vanligt namn på fjärrstyrda scripts som kör på en annan dator och sedan levererar data till användaren. Oftast använder man CGI på webservers där de levererar websidor till läsaren. CGI liknar Remote Procedure Call eller “remote shell” som finns i unix-världen.
- **Applet** är ett script som hämtas från en annan dator för att exekveras på användarens dator. Java-kod skickas oftast som applets.
- **Servlet** skickas av användaren till en annan dator för att exekveras där. Ett exempel på servlets är frågor till databaser. Servlets har vissa likheter med “remote shell”.
- **Agenter** är scripts som på ett kontrollerat sätt skickas ut bland datorerna i ett nätverk. De vandrar runt och samlar information eller letar upp en snabb dator att exekvera på och sedan kommer de tillbaka och presenterar resultatet för användaren. Agenter beskrivs närmare i kapitel 5.5. Ibland kallas de för aglets eller distribuerade scripts. Man skall dock inte förväxla dem med distribuerade objekt som förutsätts vara transparenta för användaren.

Annan litteratur om säkerhetsproblem för scripts tar endast upp applets och CGI eftersom de två metoderna har blivit vanligast. Det är en lite snäv syn som kan missa en del problem. Den här rapporten tar upp det generella problemet och försöker titta på alla former av scripts.

Det kan också vara värt att notera att lokal exekvering av lokala program är fundamentalt skild från “remote” exekvering i ett distribuerat nätverk. Skillnaden finns beskriven i [12]. Det kommer upp många fler säkerhetsproblem i applets, servlets och distribuerade scripts.

4 Säker exekvering

Det finns åtminstone fyra olika sätt att behandla mobil kod så att exekveringen blir säkrare. De är: abstinens, sandlåda, analys och omdirigering. Abstinens innebär att man försöker spärra ut all mobil kod, ofta i brandväggen (se kapitel 5.4). Man accepterar ingen mobil kod överhuvudtaget men nackdelen är att man då inte får nytta av någon mobil kod.

Sandlåda innebär att man kör koden i en skyddad omgivning där riskabla funktioner är spärrade. Den kommer att behandlas mer i detalj längre fram i det här kapitlet.

Analys innebär att man undersöker koden innan man kör den och det kan ske med matematiska och formella metoder. Tyvärr är en del programspråk, till exempel assembler svåra att analysera.

Omdirigering innebär att man dirigerar om koden till en annan dator och kör den där först för att testa den. Om den godkänns kan den köras på den första datorn eller så kan resultatet av körningen skickas istället.

Ett annat problem som inte tas upp här är det omvända problemet hur man skyddar scriptet från en server som försöker ändra i scriptet eller skickar tillbaka felaktigt resultat från körningen [5].

I det här kapitlet görs först ett försök till en beskrivning av en generell säkerhetsmodell som sedan används för att lägga en grund för sandlådemodellen. Efter det kommer ett antal programspråk och system och deras implementationer av sandlådor att undersökas och bedömas.

“The biggest problem that continues to loom on the horizon is the question of applets and downloaded executables. No solution appears to be forthcoming.” - Web Security Sourcebook [10]

4.1 Formell säkerhetsmodell

Detta är ett försök att ställa upp en generell säkerhetsmodell som kan användas för att undersöka andra specialiserade säkerhetsmodeller som till exempel sandlådemodellen. Först definieras de entiteter som ingår och sedan definieras ett antal operationer som kan utföras på dessa. Därefter ges några regler som exempel på en säkerhetspolicy.

Säkerhetsmodellen utgår från idéer i säkerhetsmodellen för aglets [4].

4.1.1 Definitioner

Definitioner av de grundläggande entiteter som ingår i modellen. De är något överlappande vilket beror på att samtidigt som de försöker täcka så mycket som möjligt så har de reducerats till så få entiteter som möjligt.

- **Subjekt** kan vara en person, process, program eller script. Det kan även vara en domän av dessa. De utför någon aktiv handling.
- **Objekt** är passiva, och är vanligen en fil med data. Det kan även vara en transaktion, en enkel text eller en I/O-enhet.
- **Kontext** är datorn, servern, operativsystemet eller interpretatorn där subjekt eller objekt kan befinna sig. Kontexter kan ha kontexter inuti sig. Det kan vara ett aktivt subjekt eller ett passivt objekt.
- **Domän** är en samling av subjekt, objekt eller kontexter. En samling subjekt kallas oftare för grupp och en samling objekt kallas i vissa system för katalog, directory, filmapp eller bibliotek.
- **Relation** visar hur enheter förhåller sig till varandra. Det kan vara en hierarki, ägarförhållande eller mer generella relationer. Ofta är det bara mellan två entiteter.
- **Kanal** är vägen mellan kontexter där objekt transporteras
- **Attribut** är något som kan ges till en entitet och därmed förse den med specifika egenskaper.

Ur dessa kan man definiera andra mer sammansatta eller specifika entiteter.

- **Policies** är en samling regler eller behörigheter. Det är alltså en domän av objekt med samma attribut. En policy kan vara ett textobjekt med attributet “policy” och flera policies kan grupperas och placeras i ett filmapp.
- **Resurs** är filsystem, skrivare, tangentbord, minne, CPU-tid, m.m.

4.1.2 Operationer

Man kan definiera ett antal primitiva operationer på entiteterna där man på något sätt ändrar deras tillstånd eller egenskap.

- **Flytta** ett subjekt eller objekt, vilket innebär att det lämnar en kontext och går in i en ny.
- **Exekvera** ett objekt. Det aktiveras och blir ett subjekt.
- **Definiera** eller omdefiniera entitet. Till exempel ett objekt görs till en kanal.
- **Skapa** en ny entitet. Till exempel ny fil med data skapas.
- **Läsa/skriva/ändra** på en entitet.
- **Radera** en entitet.
- **Tilldela** attribut.
- **Allokera** en entitet. Till exempel allokera minne.

Utgående från dessa kan man definiera fler operationer som till exempel:

- **skapa/ändra** policy eller behörigheter
- **skapa/ändra** grupper

Det finns många fler typer av operationer som man kan göra på objekt men de ligger utanför den här modellen.

4.1.3 Policies

I modellen ingår en säkerhetspolicy i form av en mängd regler. De definierar hur subjekt och andra entiteter får interagera och om hur och av vem operationerna får utföras. De måste specificera

- vilken autenticering som krävs av entiteter.
- vilka operationer ett subjekt får utföra på en entitet och under vilka villkor.
- om och hur subjekt får delegera rättigheter eller specificera egna policies.
- hur kanalerna mellan kontexter skall vara skyddade.
- vilket subjekt som är ansvarigt för varje operation.
- hur kontrollen av att policies följs eller vilket subjekt som kontrollerar att policies följs.

Varje kontext kan sedan utifrån detta lägga till fler policies. De bör vara generella och kan till exempel se ut som följande

- Varje entitet måste ha ett unikt namn.
- Om ett subjekt vill göra operationen flytta måste det autenticeras.
- Varje objekt skall ha en ägare och en tillverkare.
- Varje objekt skall ha en egen accesslista och/eller varje subjekt skall ha en capabilitylist.

Efter detta kan man specificera exakt vilka entiteter som finns, gruppera dem i domäner och sätta upp behörigheter.

4.1.4 Ett enkelt exempel

För att göra modellen tydligare så kommer här ett enklare exempel baserat på vardagen.

Ett rum har en dörr, och i rummet finns ett papper, en telefon och en dator. Rummet är en kontext, dörren är en kanal mellan rummet och resten av världen, och papper, telefon och dator är objekt. De operationer man kan göra är, läsa pappret, använda telefonen och exekvera datorn vilket innebär att man startar den. Nu kan man sätta upp policies för hur en besökare i rummet får använda prylarna, till exempel att bara ägaren får starta datorn. Man kan även autenticera alla som kommer in genom dörren med hjälp av nyckel.

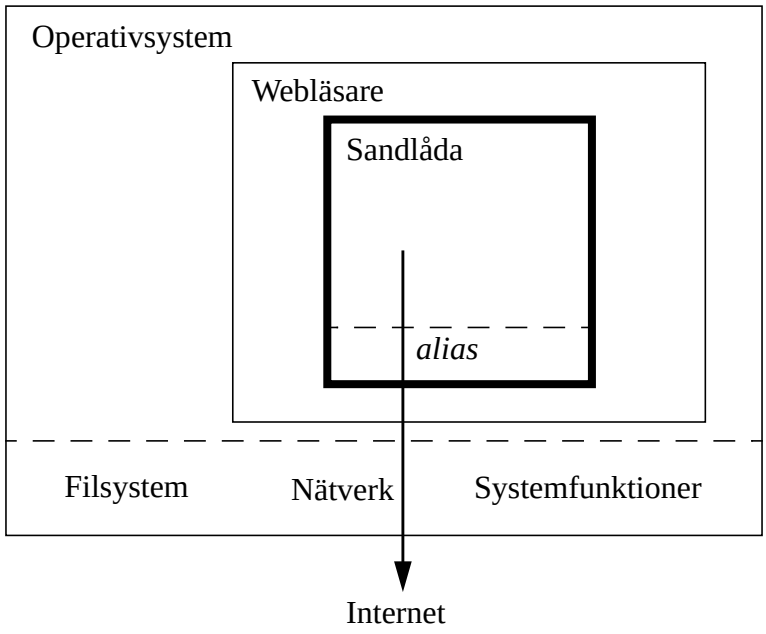
När man analyserar denna kontext ser man att rummet är ett passivt objekt som inte kontrollerar att policyn följs. Det krävs ett aktivt subjekt för detta, antingen med en vakt eller det passiva alternativet med nycklar för varje objekt. Man kan även notera att telefonen är en kanal som också behöver kontrolleras.

Ett annat exempel är inloggning på en dator. En kanal öppnas till datorns operativsystem, som då skapar ett skuggsubjekt som representerar personen. Detta subjekt autenticerar personen och startar sedan en kanal till ett annat nytt subjekt, ett skal, som sedan agerar som ombud för personen.

4.2 Sandlådemodellen

En kontext där flera funktioner och operationer är spärrade kallas för en sandlåda. Man spärrar oftast tillgång till lokalt filsystem, nätverk, bildskärm och systemfunktioner, samt begränsar tillgång till minne, hårddisk och CPU. En sandlåda har alltså en hård säkerhetspolicy.

Vanligen är sandlådan en interpretator för någon form av scriptspråk eller bytekod. Den finns då inuti ett annat program som till exempel en webläsare (webbrowser) enligt figur 1 eller som ett fristående program. När webläsaren får in ett script via en kanal så skapas en ny sandlåda och scriptet skickas in i sandlådan för exekvering. I denna skyddade omgivning får scriptet köra fritt inom lådans gränser. Om flera scripts skall köras samtidigt så får de varsin sandlåda och kommer inte åt varandra.



Figur 1. En sandlåda inuti en webläsare som kör inuti ett operativsystem. Filsystem och andra systemfunktioner anropas via alias i sandlådan. I figuren visas också en kanal till internet.

Däremot att bygga en avancerad sandlåda för maskinkod kräver ganska omfattande modifieringar av webläsare och operativsystem och är därför inte lika realistisk. Flera av de existerande operativsystemen har förstås en enklare inbyggd sandlåda.

Sandlådor liknar till stor del operativsystemskärnor vilket gör att man kan generalisera dem till en allmän säkerhetsmodell för exekvering av program.

Sandlådor kallas ibland för “padded cells”, “safe environments” eller andra liknande namn. I en del litteratur [8] ses sandlådemodellen som en variant på Trusted Computing Base.

4.2.1 Säkerhetskrav för mobila scripts

För att en sandlåda skall bli tillräckligt säker för att exekvera mobila scripts så bör vissa krav vara uppfyllda. Dessa krav är egentligen en sammanställning av krav från respektive designer av sandlådesystem. Kraven har delats upp i en ungefärlig prioritetsordning där de första är absolut nödvändiga.

1. Scriptet startar inte om det innehåller osäkra funktioner. Det görs genom analys av koden och om en detalj i scriptet bedöms som osäkert så är hela scriptet osäkert. Till exempel anrop till yttre funktioner eller om indata inte kontrolleras. Detta brukar kallas verifikation eller “taint”-regel.

2. Nätverk-, filsystem- och fönsterfunktioner spärrbara eller begränsade. Egentligen bör man kunna spärra alla funktioner.
3. Scriptet kommer inte åt yttre miljön. Till exempel får det inte läsa omgivningsvariabler eller ta reda på processortyp.
4. Minneskydd - scriptet får inte ha fri tillgång till minnet och kan bara allokera en viss mängd.
5. Identifikation - autenticering av person eller program. Detta är en förutsättning för en riktig auktorisering men det går även att köra scripts anonymt med mycket begränsad sandlåda.
6. Begränsade möjligheter att starta andra program. Det kan alltså inte skapa eller starta program utanför sandlådan, såvida det inte sker kontrollerat.
7. Auktorisering - åtkomstskydd av både filer, funktioner och nätverk. Detta är egentligen en samling regler för hur ovanstående punkter kan variera för olika personer och script. Det bör även finnas olika säkerhetspolicies att välja mellan och en kombination mellan person, programmerare och dator kan avgöra vilken policy som skall användas.
8. Begränsa tillgänglig processortid (CPU-tid), enligt något timesharing-system eller spärra funktionalitet som till exempel loopar eller funktionsdjup. Detta för att förhindra “Denial of Service” attacker som till exempel kan vara oändliga loopar.
9. Kryptera alla kanaler och alla lagrade data.
10. Säkerhetskritiska operationer loggas till något medium. Helst bör alla operationer loggas men detta kan generera för stora mängder data.
11. Säkerhetskritiska funktioner samlade på ett ställe och inte utspridda överallt i koden eller modulerna. Man får en bättre översikt om de är samlade.
12. Ett litet enkelt system som är lätt att analysera, överblicka och förstå. Detta gäller egentligen för alla typer av programsystem.
13. Ren maskinkod får inte exekveras i sandlådan utan helst bör det vara ett interpreterande språk eller pseudokod.

Ovanstående krav är lite allmänt skrivna och vissa kanske ingår i varandra men uppdelningen är gjord för att programspråken och programsystemen skall kunna jämföras.

Krav på säkra typsystem och formell bevisning för kodens korrekthet har medvetet utelämnats. Detta för att de inte visat sig fungera bättre än andra system när det gäller säkerhet. Däremot kan de vara till nytta när koden skall analyseras före exekvering eller för att upptäcka programfel på ett tidigt stadium.

4.2.2 Säkra programspråk

En del litteratur om programspråk [15] sätter upp krav på språken som enkelhet, ortogonalitet, läsbarhet men sällan något om säkerhet. Ur mina egna bedömningar har jag ställt upp några allmänna säkerhetskrav på programspråk oavsett om de skall exekveras i en sandlåda eller ej. Det finns vissa programspråk, till exempel C, som ofta är en orsak till många fel och säkerhetsproblem. Ett programspråk bör

- ha en intern och automatisk minnesallokering.
- ej ha pekare vilka man kan manipulera.
- resultera i programkod som är liten och kompakt. Ett enkelt problem skall inte resultera i flera sidor kod.
- ha en avancerad felhantering.
- ge bra och uttömmande felmeddelanden under kompileringsfasen.
- ha en öppen källkod till kompilatorn eller interpretatorn så att säkerheten i den kan analyseras av så många som möjligt.
- vara en standard som inte ägs av ett specifikt företag.
- ha ett bra typsystem som är enkelt men ändå fångar de grövsta felen. Detta för att programspråk med starka typsystem ger stor och svåröverskådlig kod.

4.3 Programspråk och system

Detta kapitel beskriver några olika programspråk och andra system, som till exempel Unix, vilka har inbyggda sandlådor eller har dem som separata moduler. Programspråken är ofta integrerade i någon webläsare som till exempel Netscape (se kapitel 5.1 om Netscapes implementation). Det finns ytterligare programspråk som kunde studeras mer ingående men de tas upp översiktligt i kapitel 5.6 istället.

4.3.1 Java

Java är ett ganska nytt språk som har blivit väldigt populärt de senaste åren. Det är utvecklat av Sun Microsystems Inc och var från början egentligen tänkt som ett programspråk för elektriska apparater. Ursprungligen hette språket “Oak” men när Sun tyckte att de behövde ett språk för World Wide Web så bytte de namn på det. Språket hade ganska tidigt inbyggd säkerhet.

Java är egentligen en förbättrad variant av C och C++, och är ett ordinärt procedurellt språk. Fördelarna med Java är att den har en intern minneshantering och saknar pekare och istället har det referenser till objekt. Minneshantering och pekare är ett stort säkerhetsproblem i C.

Innan Java-programmet kan köras måste den kompileras till bytekod som liknar den p-kod som en del Pascal-kompilatorer producerade. En virtuell maskin, Java Virtual Machine, exekverar sedan bytekoden på ett sätt som liknar interpretering. Dock finns det numera så kallade "just in time" kompilatorer som kompilerar precis innan koden körs.

Säkerhet

Förutom att Java saknar pekare och har intern minneshantering så baseras en del av säkerheten på typsystemet. Eftersom detta är ganska komplext och inflexibelt är det lätt hänt att programmeraren undviker den strikta typningen och sätter de flesta variablerna till "public" och bara har ett fåtal objekttyper, och då tappar man idén med ett säkert typsystem.

En annan viktig del av säkerheten är den så kallade säkerhetstriaden som ingår i Java Virtual Machine: Byte Code Verifier, Applet Class Loader och Security Manager. Var och en av dessa är beroende av de andra och säkerhetssystemet är beroende av att alla fungerar korrekt. Byte Code Verifiern kontrollerar att bytekoden är korrekt och har rätt format innan programmet börjar exekvera. Den använder sig av några teorier om hur bytekod bör se ut och stoppar om något ser felaktigt ut. Applet Class Loadern laddar sedan in de klasser som behövs och håller dem i en separat namnrymd. Den ser även till att klasser hämtade från nätet inte överlagrar de lokala klasserna, och den laddar klasser efterhand som de behövs. Security Managern sitter som ett gränsnitt mot en del system- och nätverksfunktioner under exekvering, vilket fungerar som en sandlåda. Om man till exempel vill öppna en socket så går anropet via Security Manager som först kontrollerar ip-adressen och som sedan anropar systemfunktionen. De här tre delarna är nödvändiga för att behålla den typsäkerhet som Java baserar sig på.

Om man med hjälp av speciella Java-kompilatorer kompilerar direkt till maskinkod så finns tyvärr inte säkerhetstriaden med. Man har dock fortfarande kvar typsäkerheten, minneshanteringen och avsaknaden av pekare. Nyare kompilatorer lär ha ett bättre säkerhetssystem.

Det finns en bok om Javas säkerhet och hur man löser en del av problemen skriven av några av experterna inom området. [3]

Kritik

Tyvärr har Java en komplex syntax, ett strikt typsystem och man måste använda objekt varje gång man vill åstadkomma något. Ett väldigt enkelt "Hello World" program blir 6 rader och inte trivialt att skriva.

```
public class HelloWorld {
 public static void main(String[] args) {
  System.out.println(“Hello world”);
 }
}
```

Dessutom är exekveringshastigheten inte särskilt snabb och även om programmet går att kompilera till riktig maskinkod så blir det ganska långsamt ändå. Anledningen till detta är att det ofta sker en dynamisk typkontroll och namnuppslagning under exekvering.

När det gäller minneshantering så allokeras arrayer dynamiskt vilket innebär att typ- och säkerhetskontroller måste göras under exekvering. Detta hamnar då utanför säkerhetstriaden, basen för säkerheten. Javas garbage collect är indeterministisk och därför vet man inte när avallokeringsfunktionen finalize() anropas.

Enligt en del experter [1] finns det, förutom ett antal buggar i tidigare implementationer, ett antal designfel i Java.

- Det finns ingen formell semantik för språket eller typsystemet. Ändå baseras säkerheten på just typsystemet.
- Ett svagt modulsystem. Det ser hierarkiskt ut i namngivningssystemet men det finns bara en nivå, dvs moduler går inte att nästla. Om man kunde det så skulle man kunna skriva ännu säkrare kod.
- Ej flexibelt accessystem till modulerna. Antingen får man full access eller ingen access. För att få en del saker att fungera så sätter den late och bekväme programmeraren full access på allt.
- Metoder kan anropas inuti konstruktorer. En metod kan alltså operera på ett partiellt initialiserat objekt.
- Bytekoden är linjär. Det gör det mycket svårare att verifiera säkerheten i bytekoden eftersom man måste göra global dataflödesanalys. Exceptions gör det ännu svårare.
- *private* modifieraren har ingen betydelse i kod från det lokala filsystemet. Kod från java.lang kan alltså sätta variabler i security managern.

I Java verkar det som om man vill dra nytta av medvinden från C och C++ genom att många programmerare redan kan de språken, men tyvärr har man nog tappat en av de största fördelarna med C; möjligheten att skriva snabb kod och få tillgång till de flesta systemfunktionerna. Som jag ser det så har man överreklamerat Java. Det är ingen revolutionerande uppfinning. Det är en bra samling nya idéer.

4.3.2 SafeTcl

Tcl är ett interpreterande scriptspråk. Det har en enkel syntax som baserar sig på att varje rad börjar med ett kommando, och det har några enkla regler för att förbehandla kommandoraden innan den exekveras. Varje kommando anropar en procedur som kan vara en intern funktion i Tcl eller en funktion i en utvidgning skriven i C eller något annat språk. Tcl behandlar alla data som strängar, saknar pekare och sköter all minneshantering internt i interpretatorn.

Säkerhet

SafeTcl är en variant av Tcl som på ett kontrollerat sätt kan exekvera scripts. Det är en master interpretator som tar emot ett script och exekverar detta i en annan separat interpretator. I denna lägger masterinterpretatorn in alias för osäkra funktioner som begränsar eller spärrar dem helt enligt någon policy. Exempel på kommandon som är begränsade är “source” - läsa in mer kod som exekveras, “open” - öppna en fil, och “puts” - skriva ut text. Nätverksfunktionen “socket”, och systemfunktionerna “cd” och “exec” är helt spärrade.

Man kan från mastern lägga in alias till systemfunktionerna med hjälp av kommandot “interp alias” eller lägga in en redan öppen I/O-kanal med “interp transfer”.

Scriptet exekveras sedan i den separata säkra interpretatorn. Inifrån interpretatorn kommer scriptet bara åt de funktioner som det blivit tillåten att anropa och inga andra yttre funktioner eller omgivning.

Det finns ett flertal olika policies. En ger tillgång till nätverket, i en annan kan man öppna lokala filer, och i en tredje får man i princip tillgång till hela den yttre miljön och andra funktioner.

SafeTcl finns som plug-in för Netscape, som beskrivs i kapitel 5.1

Kritik

I nyare versioner av SafeTcl är det tänkt att olika scripts skall få olika säkerhetspolicies med hjälp av autenticering. Tyvärr baserar sig dessa policies på autenticering med domännamn vilket gör dem känsliga för IP-spoofing [2]. Det som dock är positivt är att de har inklusive-listor istället för exklusive-listor som är osäkrare.

Eftersom Tcl behandlar all data som strängar och inte förkompilerar scriptet så blir det ganska långsamt att köra. En ny version av Tcl sägs vara snabbare. En annat problem är att det inte finns särskilt användbara metoder för att bygga avancerade datastrukturer. Dock finns det arrayer.

4.3.3 Penguin (Perl)

Perl är ett interpreterande programspråk som liknar C och shell-programspråken till Unix. Det har egen minnesallokering, ett avancerat modulsystem, möjlighet att bygga komplexa datastrukturer, och är förmodligen det språk som är bäst på reguljära uttryck. Det är egentligen mest avsett för textbehandling, vilket förstås är användbart i internet-sammanhang där mycket text skickas runt. Perl är utvecklat av en lingvist och därför ligger språkets syntax och semantik närmare naturligt språk. En nackdel är att Perl-kod ofta kan se oläslig ut när den är skriven av oerfarna programmerare men ofta resulterar det i små och kompakta program.

Säkerhet

Penguin är ett enklare sandlådesystem skrivet i Perl. I detta kan man skicka Perl-script till en annan server, krypterat och autenticerat med PGP (Pretty Good Privacy, ett paket för kryptering och autenticering), för att sedan exekveras i den andra serverns sandlåda. Beroende på användarnamn på den som skickar koden spärras olika funktioner enligt vissa policies. Om scriptet innehåller funktioner som är spärrade så startar det inte ens utan avvisas direkt. Om koden är godkänd stoppas lämpliga funktioner in i lådan och överlagrar eventuellt andra funktioner. Servern exekverar sedan koden och skickar tillbaka resultatet krypterat med PGP. I princip vilka funktioner som helst kan spärras, inklusive loopar och andra styrsatser.

Perl och även Penguin använder sig av “taint”-principen. Den innebär att indata som kommer utifrån, till exempel om de har kommit via nätverket eller lästs från filsystemet, markeras med ett “tainted” attribut. Alla variabler som sedan tilldelas dessa data som värde blir också “tainted” och markeringen propagerar vidare. För att få bort stämpeln så måste variabeln behandlas eller undersökas i programmet. Om man inte gör det får man felmeddelanden och programavbrott om variabeln används. Denna princip gör att scripten blir säkrare och detta tillsammans med den goda texthanteringen kan vara en av anledningarna till att Perl har blivit ett vanligt programspråk inom CGI-programmering.

Kritik

Tyvärr är Penguin inte färdigutvecklat än och därmed lite primitivt. Det finns inget färdigt system som kan kopplas samman med Netscape eller webservers. Det är förmodligen mer tänkt som en modul som kan vidareutvecklas och då på en väg utanför de kommersiella systemen.

4.3.4 ActiveX

ActiveX är inget programspåk utan en samling teknologier, protokoll och API:er (Application Programming Interface), som används för nerladdning av exekverbara programobjekt i ett distribuerat system som till exempel Internet. Det liknar Netscapes plug-ins med skillnaden att de laddas ner automatiskt över nätverket och tas bort när de inte längre används. Objekten kan aktivera delar av en websida, ändra menyer, läsa och skriva på hårddisken, starta andra program eller stänga av datorn. Om man bortser från den bristande säkerheten så är det ett flexibelt och användbart system. Om programtillverkaren vill uppgradera ett program så skickar de ett ActiveX objekt som gör det automatiskt.

Objekten kan vara skrivna i vilket annat programspåk som helst som är kompilerat till maskinkod. Om man vill köra Java så skickas bytekoden till en Java interpretator (JVM) som sedan exekverar koden.

Säkerhet

ActiveX använder sig av ett system för signering av objekten som de kallar Authenticode. Objekten signeras med en digital signatur och ett X.509 certifikat. Innan objekten startas så kontrolleras signaturen och om den är felaktigt så avvisas objektet. Ingen annan kontroll sker och objektet exekverar inte i någon sandlåda. Enligt Microsoft som tagit fram ActiveX är detta tillräckligt för att få säker exekvering eftersom användaren bara ska ladda ner objekt från pålitliga källor.

Kritik

Tyvärr kan ett ActiveX objekt få fullständig kontroll över en Windows 95/NT dator, när den väl är nerladdad och godkänd av användaren, och många användare godkänner obekymrat allt. Dessutom är ActiveX objekten begränsade till Microsoft världen.

4.3.5 Unix

Unix är ett gammalt operativsystem och har en del inbyggd säkerhet när det gäller exekvering. Av den anledningen tas det upp i den här rapporten och för att ha något annorlunda att jämföra med. Man kunde förstås lika gärna jämfört med Windows/NT eller VMS.

När en användare loggar in och startar ett program så exekverar det i en egen omgivning med en del spärrade systemfunktioner. Användaren kan komma åt nätverket ganska fritt men får inte tillgång till alla filer utan bara de som han har rättigheter till. Det finns privilegierade maskininstruktioner och minne som en användare inte kan komma åt.

En sak man kan notera är att man sällan hör något om virus i Unix, men däremot är de vanliga i MS-DOS. Det lär till stor del bero på den variant av sandlåda som finns i Unix-liknande system.

Kritik

En nackdel med Unix är den stora flexibilitet man har i systemet när man kan starta nya processer med pipes och kan komma åt ganska mycket av systemfilerna. Åtkomstskydd av filer är också ganska primitiv, eftersom det i princip bara finns två säkerhetsnivåer, root och användare. Den som är root kommer åt alla filer och hela systemet. Det finns också ganska begränsade möjligheter att bilda grupper av användare.

En annan nackdel är att många bakgrundsprocesser kör som root och om dessa kan fås att exekvera kod utifrån så får man lätt root privilegier. Ett annat problem är att filen som innehåller de krypterade lösenorden ofta är läsbar för alla.

4.3.6 Ytterligare exempel

Det finns andra typer av system som liknar sandlådor eller skulle behöva en sandlåda med dess accesskontroller och säkerhetspolicies. Här tas upp några exempel som dock inte kommer att tas med i jämförelsen utan enbart för att visa på paralleller.

Word makro

Det finns ett makrospråk i Microsoft Word som används för att automatisera delar av dokumenthanteringen. För att göra detta så smidigt som möjligt låter man makrona få fri tillgång till filsystem och Word självt. Det saknas alltså någon form av auktorisering med spärrade funktioner eller sandlåda.

Avsaknad av sandlåda i kombination med att det är vanligt att skicka Word-dokument på disketter eller på internet har gjort att virus skrivna i detta makrospråk har blivit ett problem. Detta visar på ett mycket konkret sätt vilka problem som kan uppkomma om man exekverar okända scripts, från okända personer i ett nätverk, i en oskyddad omgivning.

Modulsystem i programspråk

De flesta programspråk har procedurer eller funktioner. En modul är ett sätt att gruppera funktioner eller procedurer och lokala variabler. En del programspråk har ett inbyggt system för att hantera moduler. Dessa modulsystem har ofta inbyggda metoder för accesskontroll där man kan specificera vilken annan modul som kan använda den aktuella modulen. I en del programspråk kan man nästla procedurer eller funktioner så att de inte kan komma åt data utanför sin scope (räckvidd).

Modulerna har även accesskontroll för funktioner och variabler genom att ange om de är publika eller privata. Tyvärr är de flesta modulsystem ganska enkla och tillåter åtkomst antingen för alla eller ingen. De skulle förmodligen bli säkrare om de använde sig av någon variant av sandlådemodellen och någon variant av åtkomstskydd till funktioner och variabler.

Fleranvändarspel (MUD)

Det finns en slags datorspel som är en textbaserad virtuell verklighet där flera personer kan spela samtidigt. Man loggar in som en användare med namn och lösenord och kan sedan vandra runt i rummen

och utföra de kommandon som finns i rummet. Man kan till exempel plocka upp saker, släppa saker eller äta sakerna. Som en sådan vanlig användare är man begränsad till de kommandon som finns i rummet eller de objekt man bär på. Om man däremot har kodningsrättigheter så kan man skapa nya objekt genom att skriva kod för det och sedan ladda in det i spelet. Dessa objekt kan anropa funktioner i andra objekt och på det sättet manipulera omgivningen. Det har dock ett inbyggt åtkomstskydd som kontrollerar personens namn och vilka rättigheter denne har, vilket gör att man inte kan manipulera allt i spelet. Är man däremot administratör så kan man i princip manipulera allt.

Det här systemet har alltså en inbyggd sandlåda som på sätt och vis liknar ett operativsystem. Några av de här fleranvändarspelen har ett avancerat system för filskydd med hjälp av ACL:er (se kapitel 5.7).

Det intressanta med de här spelen är att man kan dynamiskt ladda in nya objekt i spelet som sedan kan interagera med spelarna. De skulle kunna fungera bra som en objektorienterad simuleringsmiljö.

Kritik

Fleranvändarspelen är för specialiserade för att användas till mobila scripts och de är ganska instabila. Dessutom finns det ingen möjlighet att komma åt nätverket.

4.4 Jämförelser mellan sandlådor

Med utgångspunkt från de säkerhetskrav som ställdes upp i kapitel 4.2.1 kan man göra en jämförelse mellan de programsystem som beskrivits ovan. Jämförelsen redovisas i tabell 1 där ett nästan uppfyllt krav markerats med • och de som uppfyller kravet helt har markerats med ••. En del extra krav som inte har med säkerhet att göra har också tagits med. Bland annat om programsystemet finns installerat i någon webläsare och om det finns i en full fungerande version och inte är experimentell. Flera av systemen utvecklas hela tiden och uppfyller i nyare versioner fler krav.

Säkerhetskrav		System				
		Java	SafeTcl	Penguin	ActiveX	Unix
1.	Verifikation	•		•		
2.	Spärrade funktioner	•	••	••		•
3.	Spärrad omgivning	•	•	•		
4.	Minnesskydd	••	•	•		•
5.	Autenticering			•	•	inlogg
6.	Ej exekvera program	•	•	•		•
7.	Auktorisering		•	•		•
8.	Begränsa CPU					
9.	Kryptering			•		
10.	Loggning					•
11.	Samlad säkerhet					
12.	Enkelt system					
13.	Ej maskinkod	•	•	•		
A.	Plug-in i webläsare	•	•		•	
B.	Fungerande version	•	•		•	•

Tabell 1. Jämförelse av programsystem.

Av denna jämförelse kan man se att för Java och SafeTcl har man satsat på fungerande system, och för Penguin ett säkrare. Det finns egentligen inget system där man kan begränsa CPU-tid förutom på en del Unix-varianter och andra operativsystem. Inget system tycks ha säkerheten samlad på ett ställe utan många har det utspritt i koden. Ej heller är de särskilt enkla att överblicka.

Man kan också se att ActiveX har jämförelsevis dålig säkerhet, men att Unix trots sin ålder har en duglig sandlåda.

4.5 Slutsatser

En slutsats man kan dra är att det finns en mängd programsystem som på ett eller annat sätt implementerar sandlådemodellen och att de uppfyller kraven mer eller mindre väl. Det finns dock inget system just nu som uppfyller hela kravlistan.

Ett system benämnt DCE har funktioner för såväl autenticering som auktorisering, och skulle kunna användas som en primitiv sandlåda. DCE beskrivs kortfattat i kapitel 5.3. Förmodligen skulle det gå att göra andra liknande modeller baserat på den formella säkerhetsmodellen i kapitel 4.1. Detta kan vara ett ämne för fortsatta undersökningar.

Kritik av sandlådemodellen

Det man inte vet är om sandlådemodellen är en tillräckligt säker exekveringsmodell. Är det en “Maginot-linje”? Vad händer om en liten detalj brister? Bryter allt samman då? Det bör kanske finnas fler försvarslinjer innanför sandlådan? Är sandlådemodellen för komplex och svår att överblicka?

Grundmodellen är egentligen ganska enkel men den blir ganska komplex om man har en stor mängd policies och system för åtkomsthantering. Det finns en intressant diskussion och kritik av sandlådemodellen på en av JavaSofts websidor [8].

Det behövs en bra och vettig säkerhetsmodell som man sedan bygger ett system utifrån. Det som händer inom området just nu är den vanliga cykliska utvecklingen idé-prototyp-test-teori-modell-ny prototyp.

Riskbedömning vs. formell bevisning

Det är vanligt inom datavetenskapen att man vill bevisa programs korrekthet. Denna bevisning är mindre lämplig inom datasäkerhet. Det går aldrig att bevisa att ett system är säkert, däremot kan man bevisa att en algoritm är korrekt.

Statiska, strikta och rigida system är lättare att göra säkra, för att det är lättare att upptäcka förändringar i dem.

5 Övrigt

Det här kapitlet innehåller en del om övriga system och implementationer. För en del områden blir det mer detaljerat och för andra nämns bara en del system för översiktens skull.

5.1 Netscape

Netscape är en av de populäraste webläsarna (webbrowsers) och har i sina senaste versioner växt till ett ganska stort programsystem. Det har nästan blivit ett operativsystem i sig, precis som för flera andra webläsare. Man kan även utöka funktionaliteten genom att installera så kallade ‘plug-ins’. När en sida laddas ner i bläddraren och om sidan har en innehållstyp som finns med i listan över plug-ins så anropas plug-inen med sidan som parameter. Den sidan kan förstås innehålla vanlig text eller exekverbart innehåll.

Det finns även säkerhetsproblem med de plug-ins man installerar i Netscape. Risken är förstås lägre eftersom man oftast installerar plug-ins från kända tillverkare. Plug-ins liknar till viss del de mekanismer som finns för “email attachments” och andra hjälpapplikationer.

Man behöver ingen plug-in för Java eftersom Netscape innehåller en Java interpretator. Samma sak gäller för Javascript som beskrivs i kapitel 5.6.

Tcl plug-in

I Netscape kan man installera en plug-in för Tcl, och med denna kan man exekvera Tcl kod inuti Netscape och visa resultatet på en sida. Det går inte att komma åt lokala filer, sockets eller andra systemfunktioner eftersom detta anses vara en säkerhetsrisk. Denna funktionaliteten är tillräcklig för animationer, spel eller mera seriösa grafiska visningar av data där användaren kan interagera med scriptet. Man kommer tyvärr inte åt Netscapes interna data, till exempel privata nycklar, och det går inte heller att använda SSL. Tcl-plug-inen använder sig av en förenklad och begränsad variant av SafeTcl.

Min kommentar: Plug-in för Tcl fungerar bra men är nog inte tillräckligt användbart.

Egna plug-ins

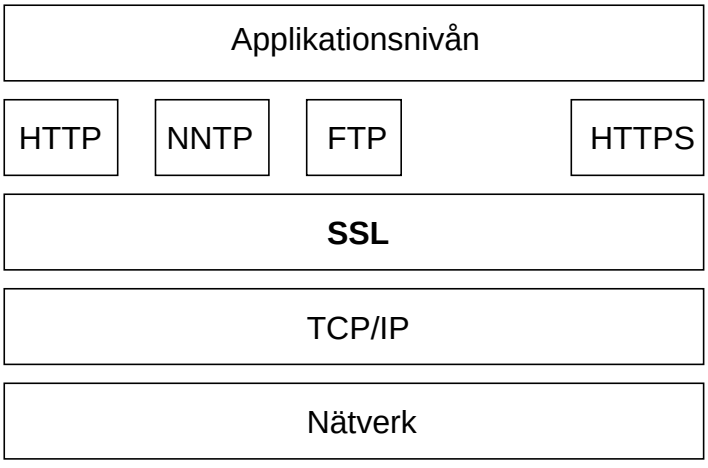
Det finns även möjlighet att skriva egna plug-ins till Netscape. Även denna möjlighet är lite begränsad vad gäller åtkomst av Netscapes interna data, men man kommer åt systemfunktioner eftersom man skriver plug-inen i C eller annat programspråk.

Egna plug-ins fungerar ungefär så här: Om Netscape hittar taggen <embed src=”source”> i ett dokument så kommer Netscape att hämta koden i “source” från servern och skicka den till plug-inen. Netscape läser sedan utdata från plug-inen och skriver ut detta i ett fönster på en sida i Netscape. Det finns lite fler funktioner i API med vilka man kan rita på sidan och liknande, men det är ändå lite begränsat.

Man skulle kunna skriva en generell plug-in som tar en source fil från en <embed> och skickar den till ett annat program via en viss port för exekvering och sedan skickar tillbaka det eventuella resultatet och visar det på en sida. Med den här plug-inen kan man sedan skriva olika typer av servers, och slipper skriva nya plug-ins för olika tillämpningar.

5.2 SSL

Secure Socket Layer (SSL) är ett protokoll framtaget av Netscape som innehåller kryptering, komprimering och autenticering. SSL är ett transparent protokoll direkt ovanför transportnivån (TCP/IP), vilket innebär att istället för vanliga socketanrop kan applikationer använda sockets via SSL. Egentligen innehåller SSL två protokoll, ett SSL Handshake Protocol och ett SSL Record Protocol där det senare ligger på samma nivå som transportnivån.



Figur 2. Protokollnivåer

När en klient som använder SSL kopplar upp sig mot en server så sker en autenticering där digitala certifikat utväxlas. Klienten och servern kan sedan kontrollera certifikatens giltighet genom att kontrollera de digitala stämplarna. I själva protokollet sker också en kontroll av de asymmetriska nycklarna genom att några slumpmässiga tecken skickas över, och om mottagaren lyckas dekryptera dem med sin privata nyckel så är denne autenticerad.

SSL kan använda sig av X.509 certifikat, i en hierarki med en Certificate Authority i toppen, eller temporära Diffie-Hellman nycklar.

SSL har nästan blivit en internetstandard och används både i Netscape och Internet Explorer, samt i ett antal webservers. Dock används inte klientautenticering särskilt ofta utan det är mest krypteringen som används. Det tas även fram en standard kallad TLS som baseras på och i stor grad liknar SSL.

5.3 DCE

DCE står för Distributed Computing Environment och är framtaget av Open System Foundation som är en grupp av dataföretagen IBM, HP och DEC. Det är ett distribuerat system som är tänkt som ett alternativ till Unix men som skall köras ovanpå andra operativsystem som VMS, Windows och även Unix. DCE innehåller ett antal tjänster, såsom trådar, Remote Procedure Call, tid, namnkataloger, säkerhet och ett distribuerat filsystem. Remote Procedure Call (RPC) innebär att en klient kan anropa en procedur i en server. En utförligare beskrivning av DCE finns i [17] och en granskning finns i [16].

Det finns tre tjänster som är intressanta. Det distribuerade filsystemet som kallas för DFS använder sig av tokens istället för frekventa kontroller som i NFS. Man begär en token när man vill skriva på en fil och dessa tokens är bara giltiga i två minuter. Om en dator går ner så kan det innebära att filsystemet “fryser” i två minuter. Detta menar man skulle lösa de problem man har i NFS. Det finns en utmärkt artikel om detta [12] som visar på att problemet är mycket svårare än så.

Det finns en säkerhetstjänst som i [17] påstås vara mycket bra. För autenticering använder den sig av Kerberos-systemet vilket innebär att lösenorden inte skickas över nätverket utan lagras i klartext på en central server. Detta ger då lite problem med att ändra eller distribuera ut nya lösenord. Om DCE hade använt sig av publika nycklar istället så skulle det fungera smidigare. Dock finns det vanlig kryptering i DCE och även digitala signaturer på meddelanden.

På samma sätt som i Kerberos finns det “tickets” som då medger att man kan göra autenticerade RPC-anrop. I de anropen kan man hämta sina privilegier från en server och sedan anropa applikationer i en

annan server. Applikationsservern anropar en ACL-server för att kontrollera användarens behörigheter. Dessa är specifika för varje resurs, men kan tyvärr inte ge ett program behörighet, endast användare eller grupper av användare.

DCE använder sig av den traditionella klient/server-modellen med RPC-anrop. Tyvärr finns inte i DCE de modernare systemen med distribuerade objekt, World Wide Web eller mobil kod.

5.4 Brandväggar

En brandvägg (firewall) är vanligtvis en dator som kör ett modifierat operativsystem som skyddar ett inre nätverk av datorer. Den blockerar en del förbindelser och släpper igenom andra, oftast en mindre mängd specificerade kanaler.

Brandväggar är egentligen bara ett skydd av kanaler, om man sätter in dem i säkerhetsmodellen. Man kan till och med säga att de är som en sandlåda fast bara skalet. Om man lyckas skicka in en Java eller ActiveX applet innanför brandväggen så kan de upprätta egna kanaler på det interna nätet eller ut på internet. Då har brandväggen förlorat sin effekt. Det är därför många brandväggar försöker filtrera bort Java och ActiveX objekt, men det kan vara svårt eftersom de kan skickas över krypterat.

Brandväggar kan spärra applets i HTML-filer genom att leta efter <applet> tagen och ersätta den med ett varningsmeddelande. De kan även spärra Java klasser genom att titta på de fyra första byten och om de är “CAFEBABE” så är det en klass. Tyvärr är det svårare att filtrera bort .ZIP-filer, .jar-filer eller andra nya format eftersom de har sina egna dataformat.

5.5 Agenter

I den populära användningen av internet så används applets ofta till att animera bilder och få andra lustiga effekter. De är dock både begränsade till websidor och till funktionalitet. För att få mer flexibilitet och användbarhet har man utvecklat något som kallas agenter. Detta kan vara små scripts som skickas runt på internet och kan till exempel samla in data, skicka meddelanden, starta program eller installera sig själv resident på andra datorer. Resultatet blir ett slags distribuerat system.

Det finns ett antal olika agentsystem. En del är ganska specialiserade för ett visst område, som till exempel databassökningar, finanssystem eller nätverkskonfiguration. Andra agentsystem innehåller lite AI och kan kommunicera med människor.

Det som är intressantast är de generella agentsystemen, till exempel AgentTcl där det är tänkt att man ska kunna skicka vilka scripts som helst. Dessa agentsystem är ofta bara prototyper och tyvärr finns det ingen säkerhet alls i dem. De liknar på sätt och vis operativsystem där agenterna kan liknas vid processer och även säkerhetsproblemen är liknande.

Det finns även ett agentsystem baserat på Java och ett som heter Telescript. Man kan även jämföra agentsystemen med distribuerade operativsystem som Spring, Mach, Amoeba eller DCE, eller jämföra dem med objektsystem som DCOM och CORBA.

5.6 Andra programspråk och system

Det finns en mängd andra programspråk som har sandlådor, eller är plug-ins eller bara är relevanta i sammanhanget. Till exempel JavaScript, Objective Caml, Castanet, ADA, Telescript, PostScript, Python, Shockwave och Inferno. Några av dessa borde studeras närmare men här tas bara upp några lite översiktligt.

Javascript

Javascript hette Livescript i början och bytte senare namn av marknadsföringskäl, fast det har inte många likheter med Java. Det är egentligen inte användbart till annat än att göra trevligare websidor. Säkerheten i Javascript består av att data markeras som “tainted” och inte kan skickas mellan olika sidor, och därmed inte mellan servrar heller. I övrigt kan man trolla som man vill med sidorna, vilket gör det användbart för Web Spoofing [2]

Min kommentar är att Javascript är inte användbart som scriptspråk i mer avancerade tillämpningar.

Objective Caml

Caml är baserat på programspråket ML och är en fransk produkt [9]. Det finns en tillhörande webläsare som kan köra Caml-applets. Säkerheten i detta systemet utgörs av pålitliga kompilatorer och digital signering av resulterande maskinkod. De pålitliga kompilatorerna finns centralt och administreras av pålitliga personer. En del av säkerheten baseras även på typsystemet i Caml.

Tyvärr ger detta samma problem som för ActiveX. Man kan inte basera säkerheten bara på digitala signaturer. Dessutom är inte ML ett särskilt lättanvänt eller populärt språk. Det är för ortogonalt, teoretiskt och vackert.

Castanet

Castanet är ett system för automatisk uppdatering av applikationer och program. Istället för “pull”-tekniken där användaren hämtar hem de uppdateringar denne behöver, så används “push”-tekniken där uppdateringarna görs automatiskt av en annan server. Man kan också se det som om man prenumererar på nya programversioner. Dessa prenumerationer kan utformas personligt för varje prenumerant.

Det saknas i princip säkerhet i systemet utan man får helt enkelt lita på distributörerna av uppdateringarna. Till skillnad från Java och ActiveX så finns det inga exekveringsbegränsningar och Castanet får obegränsad tillgång till klientmaskinen.

Den “push”-teknik som Castanet använder sig av har inte blivit särskilt populär och detta är mest på grund av att det tar mycket bandbredd på nätverken att skicka ut uppdateringar.

5.7 System för åtkomstskydd

Access Control Lists

Det finns ett system för accesskontroll som brukar kallas ACL. I detta system sätts för varje objekt (fil eller filbibliotek) en lista med identiteter och dess behörigheter. I sin enklaste form kan den se ut som i tabell 2.

Identitet	Behörighet
Adam	läsa, skriva, skapa
Bertil	läsa
Cesar	-

Tabell 2. En ACL för ett objekt

I de mer avancerade ACL systemen kan man även ge andra program behörigheter och man kan även skapa grupper av identiteter. Eftersom behörighetssystem är en av de viktigare delarna bör man lägga ner en del tid på designen. Det man kan göra är att införa prioriteter, lösenord för vissa objekt, begränsning av IP-nummer eller tidsbegränsning.

ACL:er finns bland annat i operativsystemet Solaris och även i ett MUD (se kapitel 4.3.6). I det senare fallet används det med stor framgång, förutom att användarna kan ändra i sina ACL:er själva och ibland då lyckas ta bort alla sina egna behörigheter.

Capability Lists

Dessa listor fungerar som ACL men omvänt. Istället för att räkna upp vilka subjekt som har tillgång till ett objekt, så räknar man upp vilka objekt ett visst subjekt har tillgång till. Tabell 2 visar en capability list.

Objekt	Behörighet
Fil 1	läsa, skriva, skapa
Fil 2	läsa
Fil 3	-

Tabell 3. En Capability List för en identitet

Det finns ett antal andra behörighetsmodeller som till exempel Clark-Wilson modellen eller Compacts [7]. Detta är dock tillräckligt för att fylla ytterligare en rapport.

5.8 Network Computers

Nätverksdatorer har varit på frammarsch på senaste tiden. I princip kan man likna dem vid avancerade terminaler som laddar hem kod efterhand som den behövs istället för att lagra det lokalt. Fördelarna skulle vara att det är mycket enklare att uppdatera program eller byta ut en NC.

En nackdel med att skicka kod över nätverk är att det är mycket långsammare än att hämta den lokalt. Ett annat problem är att man blir alltför beroende av nätet för att det ska fungera.

Något som man kanske kommer att få se på NC är så kallade network user interfaces (NUI). Det är i princip ett fönstersystem som är en enda internet webläsare, och denna ersätter i princip fönstersystemet och operativsystemet. Den hanterar email, news, och websidor.

5.9 Framtiden

Om framtiden är det svårt att sia men att dynamisk och mobil kod hör till den är ganska säkert. Agenter vandrar runt i datorsystem och nätverken och samlar in information. Det kommer mer eller mindre att se ut som en kopia av verkligheten. Agenterna kommer till stor del vara skrivna i något scriptspråk. De kompilerande programspråken kommer att bli allt ovanligare och nästan bara finnas i tidskritiska system.

Vi kanske kommer att få se plug-ins som installerar sig automatiskt, självuppdaterande kod, och kanske kommer brandväggar att försvinna och ersättas med flera barriärer runt varje dator eller program.

Smartcards kommer att finnas i var mans ägo. De digitala certifikaten kommer sättas upp i en hierarki eller “web of trust”, och både personer och program kommer att ha certifikat. Och samtidigt vill man kunna garantera en persons anonymitet, vilket är ett svårt problem eftersom andra personer och företag vill kunna identifiera personen.

Ett annat område som kommer att växa är insamling och sammanställning av data, “data mining and collating”. Det är inom detta område som agenterna kommer att göra nytta.

6 Referenser

[1] Drew Dean, Edward Felten, Dan Wallach, *Java Security: From HotJava to Netscape and Beyond*, IEEE Symposium on Security and Privacy, 1996

[2] Edward Felten, Dirk Balfanz, Drew Dean, Dan Wallach,*Web Spoofing: An Internet Con Game*, Technical Report 540-96, Department of Computer Science, Princeton University, Feb. 1997

[3] Gary McGraw, Ed Felten, *Java Security: Hostile Applets, Holes and Antidotes*, ISBN 0-471-17842-X, 1997

[4] Günter Karjoth, Danny B. Lange, Mitsuru Oshima, *A Security Model for Aglets*, IEEE Internet Computing July/August 1997

[5] Li Gong, *Survivable Mobile Code Is Hard to Build?*, DARPA Workshop on Foundations for Secure Mobile Code, http://www.cs.nps.navy.mil/research/languages/wkshp.html

[6] Bruce Scheier, *Applied Cryptography*, ISBN 0-471-11709-9, 1996

[7] Martin Röscheisen and Terry Winograd, *A Communication Agreement Framework for Access/Action Control*, http://diglib.stanford.edu/rmr/SP/Framework.html

[8] JavaSoft FORUM, http://java.sun.com/forum/securityForum.html

[9] François Rouaix, *Objective Caml and MMM browser*, http://pauillac.inria.fr/~rouaix/mmm/

[10] Aviel D. Rubin, Daniel Geer, Marcus Ranum, *Web Security Sourcebook*, ISBN 0-471-18148-X, 1997

[11] Simson Garfinkel, Gene Spafford, *Practical Unix & Internet Security*, ISBN 1-56592-148-8, 1996

[12] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall, *A Note on Distributed Computing*, Sun Microsystems Laboratories Inc Technical Report TR-94-29

[13] H Säk IT, Handbok för Försvarsmaktens Säkerhetsskyddtjänst Infomationsteknologi, 1996

[14] Per Svensson mfl., *Ny systemarkitektur för evolutionär utveckling av lednings- och informationssystem för försvaret*, FOA-R--98-00673-505--SE

[15] C Ghezzi, *"Language Design", Chapter 10, Programming Language Concepts*, ISBN 0-471-85399-2, 1987

[16] *Special Distributed Computing Report*, Byte, June 1994, p 121-206

[17] Andrew S. Tanenbaum, *Distributed Operating Systems*, ISBN 0-13-219908-4, 1995