Säker Java kod en kvalitativ studie

Abstrakt

Datasäkerhet är ett omfattande ämne som blir allt viktigare pga. att världen allt mer sammanlänkas, genom nätverk som sköter kritiska transaktioner. Miljön i vilken datorer måste överleva har förändrats radikalt sedan populariseringen av Internet. Att ansluta ett lokalt nätverk (LAN) till Internet är ett säkerhetskritiskt beslut. Källan till de flesta säkerhetsproblem är mjukvara som fallerar på oväntade sätt. Därför blir det idag allt mer självklart att tyngdpunkten bör läggas på att säkerhetsmässigt förbättra den kod som skrivs. Uppsatsen inriktar sig på att skildra de vanligaste misstag som görs säkerhetsmässigt samt olika programmeringstekniker för att förbättra koddesignen vid skrivande av Java kod. Studien baserar sig på akademisk facklitteratur och Internetartiklar inom ämnet. Resultatet av studien visar att många misstag begås som kan ge upphov till säkerhetshål vid skrivande av Java kod. Bl.a. överskattas ofta de inbyggda säkerhetsmekanismer som följer med högnivåspråket. Genom att uppdaga de vanligaste misstag som görs och diverse programmeringstekniker är det möjligt att förbättra koddesignen. Detta leder i sin tur till säkrare applikationer och bättre säkerhet i allmänhet. Slutsatserna av studien är att det är möjligt att avsevärt förbättra den säkerhetsmässiga designen av Java kod, genom att undvika dylika fallgropar inom områdena åtkomst, serialisering, paket, privilegierad kod och inhemska metoder. Dessa områden bedöms som extra känsliga när det kommer till säker Java kod och blir ofta förbisedda av orutinerade programmerare.

Avser: C-uppsats, 10p
Ämne: Informatik
Handledare: Andreas Nilsson
Författare: Daniel Susid
Datum: 2003-01-03

Inledning

Säkerhet är alltid av yttersta intresse för utvecklare, eftersom den utgör en oerhört väsentlig del av en applikations kvalitet (Garms & Somerfield, 2001). Den stora tillväxten av Internetanvändande har dessutom ökat människors medvetenhet om nödvändigheten av säkra applikationer och de problem som uppstår vid misslyckande inom detta område. För dagens företag innebär därför luckor i datasäkerheten inte enbart en intern pinsamhet, utan är något som kommer att minska människors förtroende för företaget. Företag som arbetar med t.ex. e-handel eller B2B har inte råd med sådana säkerhetshål, som skulle kunna innebära en förlust av kunder, förstörda arbetsrelationer och partnerskap vilket i sin tur kan äventyra företagets hela framtid.

Framtill idag har tyngdpunkten i olika företags och andra organisationers datasäkerhet lagts på installation av brandväggar och Virtuella Privata Nätverk (VPN) för att hindra utomstående crackers dvs. individer som försöker bryta sig in i ett system med skadliga avsikter, från att komma åt värdefull företagsdata (Nanjunda, 2002). Men enligt en tidigare publicerad rapport (CSI/FBI, 2002), är mer än 30 procent av alla företagssäkerhets hål interna. Med andra ord innebär företagsapplikationer ett mycket sårbart säkerhetsområde som måste adresseras av IT-styrningen. Av denna anledning vänder sig företag idag allt mer till industricertifieringar för att försäkra sig om att deras applikationer uppnår de krav som krävs, inklusive säkerhetsstandarder för att skydda deras dyrbara data/information (Nanjunda, 2002).

De vanligaste säkerhetsaspekterna vid skrivande av säkra applikationer är (Garms & Somerfield, 2001):

- Vilket programmeringsspråk är mest lämpligt för att uppnå den säkerhetsnivå som önskas?
- Vilket operativsystem är mest lämpligt för att uppnå den säkerhetsnivå som önskas?
- Hur skrivs kod som inte ger upphov till säkerhetshål?
- Hur överförs känslig information som kreditkortsnummer?
- Hur lagras känslig data?
- Hur verifieras att kod är från en pålitlig källa?
- Hur förhindras obehöriga från att få tillgång till systemet?

Det första men kanske även det viktigaste som bör tänkas på när det kommer till datasäkerhet i allmänhet men gäller likaså säkra applikationer, är att det inte finns något som kan kallas helt säkert (Lange, 1997). Allt kan hackas på ett eller annat sätt och det finns inga undantag till denna regel. Vad en programmerare kan göra är att försvåra processen för en attackerare och göra det enklare att sedan restaurera det som blivit kränkt säkerhetsmässigt (Garms & Somerfield, 2001).

En säker applikation skall kunna klara av att stå emot vissa typer av attacker som bl.a. DoS (Denial of Service), virus, avlyssning, otillåten åtkomst och manipulering av data (Garms & Somerfield, 2001). Detta kan uppnås genom att man undviker att skriva kod som kan ge upphov till kända säkerhetshål samt använder sig av kryptering, autentisering och mekanismer för oavvislighet. Självklart måste även den underliggande arkitekturen vara säker, annars spelar det ingen roll hur säker en applikation är. Säkerheten måste utgöra en del av utvecklingsprocessen och inte enbart vara en eftertanke. När det kommer till säkerhet skall

applikationen beaktas från slut till slut och där leta efter sårbarheter i alla olika punkter i arkitekturen.

När Java först släpptes av Sun Microsystems, blev det uppmärksammat av programmerare runt hela världen (Oaks, 2002). Dessa utvecklare blev intresserade av Java pga. många olika skäl. En del attraherades av dess plattformsoberoende egenskaper, andra av dess enkelhet i jämförelse med andra objektorienterade språk som t.ex. C++, andra av dess stabilitet och minneshantering, andra av dess säkerhet och andra av ytterligare skäl. Precis som utvecklare drogs till Java med olika förväntningar, hade de med sig olika förväntningar om vad som menades med det myntade uttrycket ”Java är säkert”. Säkerhet betyder olika saker för olika människor och många utvecklare som hade vissa särskilda förväntningar om ordet ”säkerhet” blev förvånade att upptäcka att deras förväntningar inte nödvändigtvis delades av Javas designers.

Ett vanligt och alarmerande fenomen när det kommer till Java och säkerhet som beskrivs av Sundsted (2001) i en Internetartikel, är att många utvecklare arbetar under missuppfattningen att de inte behöver bry sig om säkerhet på grund av att ”Java är säkert”. Enligt Sundsted (2001) gör utvecklarnas acceptans av den felaktiga bilden, att de inte inser att de måste de betrakta säkerheten ur tre olika perspektiv:

- Javatolksäkerhet
- Applikationssäkerhet
- Nätverkssäkerhet

Sundsted (2001) påpekar dessutom att det är endast ett område som skulle kunna sägas vara direkt automatiskt säkert och det är Javatolksäkerheten.

En oerhört väsentlig del inom processen att skapa en säker applikation, utgörs av hur säker själva koden är (Howard & LeBlanc, 2002). Kodens säkerhet utgör själva stommen i applikationssäkerheten och måste beaktas med största tillförsikt. Bland säkerhetsexperter är det allmänt välkänt och accepterat att det inte finns något sådant som fullständigt säker kod, som är helt fri från alla möjliga säkerhetshål (Viega & McGraw 2002). Detta stämmer för de flesta komplexa applikationer, på grund av att det helt enkelt finns för många olika faktorer att beakta för att någonsin kunna vara helt säker på att alla möjliga attack vägar är stängda. De flesta programmerare har istället som mål att skriva kod som är fri från alla kända säkerhetshål och som är osannolik att innehålla nya (Garms & Somerfield, 2001).

När det kommer till säker Java kod finns det fem specifika områden som kräver extra uppmärksamhet (Sun Microsystems, 2000):

- Åtkomst
- Serialisering
- Paket
- Privilegierad kod
- Inhemska metoder

Dessa fem områden utgör alla väsäntliga delar av det som karaktäriserar språket Java och innebär sådan funktionalitet att om de används på ett ogenomtänktsätt, skapas omständigheter som ger upphov till potentiella säkerhetshål (Sun Microsystems, 2000). Det är mycket vanligt att oerfarna programmerare missar dessa områden pga. okunskap eller slarv, vilket gör att applikationssäkerheten blir lidande (Garms & Somerfield, 2001).

Syfte

Syftet med uppsatsen är att skapa förståelse för hur säker kod i Java skrivs.

Uppsatsen vänder sig främst till dem som arbetar med mjukvarukonstruktion, men kan även vara av intresse för personer som är involverade i införandet och tillämpandet av olika mjukvarulösningar.

Problemformulering

Hur skrivs säker kod i Java med avseende av dess mest säkerhetskritiska områden?:

- Åtkomst
- Serialisering
- Paket
- Privilegierad kod
- Inhemska metoder

Dessa fem områden har studerats genom kartläggning av deras inverkan på Java kod ur ett säkerhetsmässigt perspektiv.

Avgränsning

Inom området av datasäkerhet togs beslutet att skriva om applikationssäkerhet och där begränsa sig till att endast behandla hur säker kod i programmeringsspråket Java skrivs. För att skildra en lämplig riktlinje för hur säker kod i Java skrivs, behandlas de vanligaste säkerhetsmisstagen samt de områden som bör fokuseras på ur säkerhetssynpunkt. Dessutom valdes att inte ta upp andra områden rörande säkerhet och Javaapplikationer som Javatolksäkerhet eller nätverksäkerhet. Uppsatsen behandlar inte häller de olika verktyg och andra hjälpmedel som finns för att stödja applikationer säkerhetsmässigt, eller metoder för att skydda data samt styrka identiteter som kryptering och autentisering. Fokus läggs istället på hur en kod mässigt säker applikationsgrund byggs upp.

Begreppsdefinitioner

Bytekodverifieraren

Bytekodverifieraren används av Javatolken (Garms & Somerfield, 2001). Den har som uppgift att se till att endast legitim kod exekveras (Gong, 1999). Detta sker genom att den kontrollerar att bytekoden följer Javas språkspecifikationer och att det inte finns några överträdelser av Javas språkregler eller namnutrymmesrestriktioner.

Bytekod

En Javakompilator översätter Javaprogram till en plattformsoberoende bytekod representation (Gong, 1999). En Java .class fil består av bytekod för en specifik klass.

Byteström

En ström är en sorts kommunikationsväg för data från en källa till mål (Skansholm, 1999). Om data flödar in i ett program kallas det för en inström och om data istället flödar ut ur

programmet kallas det för en utström. Byteströmmar är en av två olika kategorier av strömmar. I byteströmmar överförs data i form av bytes, dvs. 8 bitar.

Javatolken

Javatolken är ett speciellt program som ingår i den virtuella Javamaskinen (Skansholm, 1999). Programmet läser Java bytekoden som körs och ser till att de instruktioner som finns i den utförs.

Javasäkerhetsmodellen

För det mesta är Javasäkerhetsmodellen uppdelad i två delar (Gong, 1999). Den ena består av den användarjusterbara säkerhetshanteraren som kontrollerar olika API operationer som tex. fil access och den andra av bytekodverifieraren som försäkrar validiteten av den kompilerade bytekoden.

Javasäkerhetshanteraren

Säkerhetshanteraren är ett enstaka Javaobjekt som utför runtime kontroller av farliga metoder (Gong, 1999). Kod i Javabiblioteket konsulterar säkerhetshanteraren när en potentiellt farlig operation skall utföras. Säkerhetshanteraren kan stoppa operationen genom att generera ett säkerhetsundantag.

JAR filer

JAR filer är samma som ZIP filer, förutom att de även har en META-INF/MANIFEST.MF fil (Gong, 1999). Med hjälp av JAR filer är det möjligt att paketera klass filer, bilder, ljud och annan data i en enda fil för snabbare och enklare distribution.

JavaBeans

JavaBeans är återanvändbara komponenter som fungerar likt en ”svart låda”, kapabel att utföra någon funktion (Bakharia, 2001). Den svarta lådan kan användas utan att veta hur den egentligen utför en uppgift. Det ända som krävs är att förse den med input parametrar för att outputen skall beräknas.

Operativsystemets paging mekanism

En dator kan exekvera program som är större än dess fysiska minne (Clements, 2002). Detta sker med hjälp av virtuellt minne. I ett virtuellt minnessystem ser programmeraren ett stort fält av vad som verkar vara snabbt primärminne. Men i själva verket består det av en mindre del snabbt primärminne och en större del långsamt diskutrymme. Detta sker genom att operativsystemet swapar ut delar av program som inte används till disk. När de sen används läses de tillbaka till primärminnet från disk. Den information som swapas från primärminnet till disk är uppdelad i enheter som kallas ”Pages”.

Primitiver

Javas virtuella maskin hanterar åtta primitiva datatyper (Harold, 1997). Dessa är byte, short, int, long, float, double, char och boolean.

Skräpsamlaren

För att inte det tillgängliga minnet skall ta slut måste utrymmet för objekt som inte längre används återlämnas (Skansholm, 1999). I Java sker detta automatiskt med hjälp av skräpsamlaren.

Virtuell Javamaskin

Den virtuella Javamaskinen är en tänkt dator som förstår den bytekod som produceras av en Javakompilator (Skansholm, 1999).

Metod

En metod är ett redskap för att lösa problem och komma fram till ny kunskap (Holme & Solvang, 1997). Allt som kan bidra till att uppnå dessa mål är en metod. Men det är viktigt att vara införstådd med att det inte betyder att alla metoder är lika hållbara eller tål kritisk prövning lika bra. Metoder som skall användas i ett forsknings- och utvecklingsarbete måste bland annat uppfylla följande grundkrav (Hellevik, 1980):

- Det måste finnas en överensstämmelse med den verklighet som undersöks
- Ett systematiskt urval av information måste göras
- Informationen skall kunna utnyttjas på bästa sätt
- Resultatet ska presenteras på sådant sätt att andra kan kontrollera och granska hållbarheten
- Resultatet ska möjliggöra ny kunskap och medvetenhet för att detta skall kunna leda till fortsatt forsknings- och utvecklingsarbete och till ökad förståelse

Normalt skiljs två olika metodiska angreppssätt åt (Holme & Solvang, 1997):

- Kvalitativa
- Kvantitativa

Detta görs med utgångspunkt från den information som undersöks dvs. om det är mjukdata eller hårddata. Den mest väsentliga skillnaden mellan dessa två metoder är hur siffror och statistik används. Båda metoderna har sina starka och svaga sidor. Av denna anledning är det viktigt att metodvalet görs med detta i åtanke och med utgångspunkt i den frågeställning som skall undersökas.

Kvalitativa metoder

Kvalitativa metoder innebär en ringa grad av formalisering (Holme & Solvang, 1997). De har främst ett förstående syfte. De inriktar sig inte på att pröva informationens generella giltighet. Istället utgörs deras centrala del av att genom olika sätt att samla information göra det möjligt att få en djupare förståelse av de problem som studeras och att kunna beskriva helheten av det sammanhang som dessa inryms i. Metoderna kännetecknas av närhet till den källa där informationen hämtas.

Kvantitativa metoder

Kvantitativa metoder är mer formaliserade och strukturerade (Holme & Solvang, 1997). Metoderna präglas i större utsträckning av kontroll från forskarens sida. De definierar vilka

förhållanden som är av särskilt intresse utifrån de frågeställningar som valts. Metoderna avgör också vilka svar som är tänkbara. Deras uppläggning och planering kännetecknas av selektivitet och avstånd i förhållande till informationskällan. Allt detta är nödvändigt för att det skall gå att genomföra formaliserade analyser, göra jämförelser och pröva om de resulterande resultaten gäller alla de enheter som det är önskvärt att uttala sig om. Statistiska mätmetoder spelar en central roll i analysen av kvantitativinformation.

Val av metod

I denna studie har en kvalitativ metod använts. Den kvalitativa metoden är den mest lämpliga för att undersöka bakomliggande faktorer och förstå hur saker och ting fungerar. Det finns ett behov av att nå en djupare insikt inom problemområdet. Vad som orsakar vad och varför det gör det. Av denna anledning fokuserar studien till högre grad på djup än bredd.

Använd metod

En teoretisk litteratur studie genomfördes och baseras på två typer av källor: Internetartiklar samt akademisk facklitteratur. Den teoretiska litteraturstudien bestod av att samla in litterärt material i syfte av individuella självstudier. Genom studien inhämtades den kunskap om problemområdet som krävdes, för att framställa det resultat som presenteras i uppsatsen.

Den sökmotor som använts mest vid litteratursökningen på Internet är www.google.com. De flesta av artiklarna som används i uppsatsen hittades med hjälp av Google. Sun's hemsida www.javasoft.com har också varit till hjälp, genom dess tillhandahållna artikeldatabas. Även www.amazon.com användes för att söka akademisk facklitteratur inom området.

De sökord som använts vid sökning av akademisk facklitteratur och artiklar är: java security, secure java programming, secure java applications, secure java code, java, security. Majoriteten av dessa begrepp har uppdagats genom tidigare litteraturstudier. Valet att söka på dessa begrepp gjordes pga. att de starkt knyter an till det aktuella ämnesområdet.

Validering av teori

När det kommer till validering av teori är det viktigt att vara selektiv och använda sig av så aktuella uppgifter som möjligt. Även om området av säker kod i allmänhet inte förändras i lika extremt tempo som resten av datasäkerhetsområdet, är det väsentligt att leta efter uppgifter som är tidsenliga. Självklart kan dessa uppgifters värde inte alltid bedömas efter en tidsskala, eftersom det ofta kan finnas äldre uppgifter som är intressanta ur ett IT-historiskt perspektiv. Säker kod är trots allt ett område där lärdomar dras från mer erfarna programmerares tidigare misstag, som ofta är tidlösa.

Resultat

Säker Java kod

Javasäkerhetsmodellen är designad för att skydda användare mot farlig kod, men det är möjligt att skriva kod som oavsiktligt tillåter annan kod att kringgå en del av dessa säkerhetsmekanismer (Garms & Somerfiled, 2001). Eftersom en del kod arbetar med känslig information och resurser som filer och lösenord, är det viktigt att mjukvaran som skrivs inte

innehåller några hål som skulle göra det möjligt för någon att komma åt dessa resurser utan tillstånd.

När det kommer till Java är det möjligt att känna igen och undvika många av de olika kodningsmisstag som ger upphov till säkerhetshål, samt ta bort dem innan de utgör något problem (McGraw & Felten, 1998). Följande områden är särskilt viktiga att ha i åtanke för att undvika säkerhetshål vid skrivande av kod i Java (Sun Microsystems, 2000):

- Åtkomst
- Serialisering
- Paket
- Privilegierad kod
- Inhemska metoder

Åtkomst

Synlighet

Vid skrivande av Java kod är det viktigt att göra rätt beslut om hur synligheten för metoder, klasser och medlemsvariabler deklareras (Sun Microsystems, 2000). De fyra olika synlighetsnivåerna i nedstigande ordning är (Skansholm, 1999):

- Public – Det som deklareras som public är synligt och tillgängligt överallt i programmet, även från andra klasser.
- Protected – Det som deklareras som protected är endast tillgängligt i den aktuella klassen, i andra klasser i det aktuella paketet samt i eventuella subklasser i andra paket.
- Paket – Blir tillgängligt i alla andra klasser i det paket som den aktuella klassen ingår i, men aldrig från några klasser som ingår i några andra paket. Denna synlighetsnivå är automatiskt förvald och används när ingen annan synlighet deklarerats.
- Private – Det som deklareras som private är endast tillgängligt i den aktuella klassen.

Antagande att bytekodverifieraren är påslagen, kommer det vara omöjligt att nå begränsade fält och metoder (Garms & Somerfield, 2001). Därför bör man sträva efter att reducera dessa entiteters synlighet så mycket som möjligt (Sun Microsystems, 2000). Om en metod endast behöver anropas inifrån en enda klass, borde den metoden deklareras som private (McGraw & Felten, 1998). Likaså om en variabel endast behöver kommas åt inifrån en enda metod utan någon fortlevnad, borde den vara en lokalvariabel och inte en medlemsvariabel.

Medlemsvariabler skall deklareras som private när det är möjligt, där åtkomsten sker via accessmetoder eller som de också kallas set- och getmetoder (Garms & Somerfield, 2001). En klass som använder sig av accessmetoder för att skydda en private deklarerad medlemvariabel kan se ut som följande exempel:

```
public class Account
{
   private float accountBalance;
   public float getAccountBalance()
   {
```

```
        return accountBalance;
    }
    public void setAccountBalance( float accountBalance )
    {
        this.accountBalance = accountBalance;
    }
}
```

Detta är standard Java praxis för hur JavaBeans skrivs och detta av ett bra skäl (Garms & Somerfield, 2001). Om säkerhetskontroller behövs kan de centraliseras i dessa accessmetoder, istället för att koda om alla klasser som använder sig av dessa medlemsvariabler. Det är också möjligt att endast ha en set- eller getmetod, vilket möjliggör variabler som endast kan läsas från eller endast kan skrivas till.

Synlighet och säkerhet

Ett intressant fenomen som beskrivs i en Internetartikel skriven av Mogasale (2000) är att bytekodverifieraren inte är förvald för klasser som laddas från det lokala filsystemet. För att få Javatolken att använda sig av bytekodverifieraren krävs att de odokumenterade argumenten –verify eller –Xverify:all används. Ur säkerhetssynpunkt hade det varit mer förståeligt om bytekodverifieraren hade varit förvald och istället kunde stängas av om så önskades. Ett exempel på problem som kan uppstå pga. av detta fenomen är t.ex. en bugg som beskrivs av Mogasale (2000). Mogasale (2000) skriver att det går att komma åt privat data med hjälp av en riggad kompilator, direkt editerande av bytekoden eller med en enkel om kompilering. Ta t.ex. de följande två klasserna:

```
public class Outside
{
    public static void main( String[] args )
    {
        Inside inside = new Inside();
        inside.print();
        inside.value =  42;
        inside.print();
    }
}
public class Inside
{
    private int value = 23;
    private void print()
    {
        System.out.println( value );
    }
}
```

Det är ganska självklart att klassen Outside inte borde gå att kompilera eller köra eftersom den inte kan komma åt den private deklarerade variabeln i klassen Inside, vilket den självklart inte gör heller (Mogasale, 2000). Men det är relativt enkelt att lura kompilatorn att kompilera klassen och den virtuella maskinen att köra den. Genom att deklarera om variabeln value och metoden print till public och kompilera de båda klasserna, är det möjligt att få kompileringen att gå igenom och den virtuella maskinen att köra dem vilket är i sin ordning. Men om variabeln value och metoden print sedan ändras tillbaka till private. Kompilerar om klassen Inside och inte klassen Outside. Kommer klassen Outside fortfarande komma åt variabeln value och metoden print, trots att de nu åter deklarerats som private. Detta borde inte ske utan klassen Outside borde kasta undantaget IllegalAccessError istället.

Även om det ovanstående exemplet kräver tillgång till källkoden är det inte mycket svårare att utföra det utan källkod (Mogasale, 2000). Detta är ett allvarligt säkerhetsproblem då det räcker att en person har lite kunskap om bytekod hackning för att kunna öppna dörren till en rad olika attacker. Applets lider inte av detta problem då bytekodverifieraren är automatiskt förvald när de körs. För att undvika denna säkerhetsrisk i vanliga applikationer och begränsa dem med Javas säkerhetsmodell kan det vara en bra regel att använda sig av –verify argumentet för att se till att Javas åtkomstregler följs. För att köra det tidigare exemplet med påslagen bytekodverifierare skrivs: java –verify Outside.

Final klasser, metoder och variabler

Final nyckelordet deklarerar klasser, metoder och variabler som konstanter (Skansholm, 1999). Det innebär att entiteter som deklareras som final alltid förblir oförändrade. Med andra ord är det omöjligt att skapa subklasser av klasser som deklarerats som final eller överskugga finalmetoder samt tilldela nya värden till finalvariabler.

Det är viktigt att uppmärksamma att trots att en final variabel inte kan tilldelas något nytt värde, betyder det inte att den inte kan modifieras (Garms & Somerfield, 2001). T.ex. om en Hashtabell h deklarerats som final, är det omöjligt att förändra variabeln h så att den pekar på en ny Hashtabell. Men det är fullt möjligt att använda sig av Hashtabellens put() metoder. Det samma gäller för fält. En final variabel som pekar på ett fält kan alltså inte tilldelas ett nytt värde men innehållet i fältet kan förändras som önskas.

Genom att deklarera en klass som final och se till att det inte finns något sätt att förändra dess innehåll, t.ex. genom att inte ha några set() metoder, går det att skapa objekt som är oföränderliga (immutable) som String (Garms & Somerfield, 2001).

Static fält

Variabler som deklareras som static kallas ofta för klassvariabler och är tilldelade en speciell klass snarare än ett objekt (Skansholm, 1999). Det finns endast en instans för en given klass, oavsett hur många objekt som skapas av klassen. Klassvariablerna används ofta för att definiera konstanter inom en klass genom att lägga till modifieraren final (Sun Microsystems, 2000). När static deklarerade variabler behöver deklareras som public bör de även deklareras som final så att de inte kan modifieras. Annars finns det risk för att en klass beteende skulle kunna modifieras av utomstående kod. Om det krävs en offentligt åtkomlig static deklarerad variabel som behöver kunna modifieras, är det bäst att deklarera den som private och sedan returnera en kopia av den via en get() metod samt skapa en set() metod så den även kan modifieras (Garms & Somerfield, 2001). På så sätt kan modifieringar skötas inifrån klassen.

Rensa känslig information

När känslig information lagras i ett objekt, är det viktigt att känna till vad som händer med den informationen när objektet inte längre används (Garms & Somerfield, 2001). Normalt frigör skräpsamlaren det minnet för senare användande, men det finns ingen garanti att minnet kommer att rensas förrän det verkligen återallokeras av en annan begäran för mer minne (Sun Microsystems, 2000). I vissa situationer kan det vara fullt möjligt för någon att läsa från det minnet efter att applikationen har avslutats eller under dess själva exekvering, om det finns något sätt att läsa från minnet direkt under det operativsystem som applikationen körs på (Viega & McGraw, 2002).

T.ex. för ett privat nyckel objekt som lagrar den privata nyckel som används vid dekryptering i assymetriskkryptering, skulle klassen kunna se ut på följande sätt (Garms & Somerfield, 2001):

```
public class PrivateKey
{
   private byte[] keyContents;

   // Metoder här….
}
```

Sedan när den privata nyckeln används för att dekryptera ett meddelande och den virtuella maskinen efteråt avslutas, kan en C applikation allokera en stor bit minne som söks igenom (Garms & Somerfield, 2001). På så sätt kan det vara fullt möjligt att finna den privata nyckeln, som är ett fält av bytes.

Idealet när det kommer till hantering av känslig information, är att rensa bort datan när den används färdigt (Sun Microsystems, 2000). I PrivateKey klassen är det möjligt att lägga till en clear() metod som gör just detta, som tex. i det följande exemplet (Garms & Somerfield, 2001):

```
public class PrivateKey
{
   private byte[] keyContents;

   public void clear()
   {
      for ( int i = 0; i < keyContents.length; i++ )
      {
         keyContents[ i ] = ( byte ) 0x00;
      }
   }
   //  Andra metoder här….
}
```

Med hjälp av clear() metoden är det nu möjligt att nollställa byten som utgör keyContents byte fältet, så att ingen annan kan läsa dem (Garms & Somerfield, 2001). Beroende på ens underliggande datastrukturer kan det ibland vara svårt att rensa objekt. Det är väldigt viktigt

att uppmärksamma att det inte räcker att bara sätta dem till null. Utan det är nödvändigt att gå ner på den nivå som primitiverna är lagrade och sätta dessa till nollvärden.

För att försäkra sig om att den känsliga datan blivit rensad måste clear() metoden anropas när t.ex. PrivateKey använts klart (Garms & Somerfield, 2001). Det går att anropa clear() från finalize() metoden, men det garanterar inte att minnet kommer att rensas innan den virtuella maskinen avslutas. Det är mycket bättre att anropa clear() manuellt.

Oföränderliga (immutable) klasser

En del klasser i Java är oföränderliga (immutable), som t.ex. String och BigInteger (Sun Microsystems, 2000). Eftersom de inte kan förändras, kan de heller inte rensas bort. Lösningen i det här fallet vore att helt enkelt inte använda sig av String eller BigInteger för att lagra data som behöver kunna nollställas (Garms & Somerfield, 2001). Istället skall lösenord och privata nycklar lagras i fält och ens klasser förses med clear() metoder för dessa fält.

De flesta kryptografiska implementeringar i Java lagrar sina privata nycklar i BigIntegers (Garms & Somerfield, 2001). Detta medför att det är teoretiskt möjligt för någon att söka igenom minnet för att finna privata nycklar, eftersom det inte finns något sätt att nollställa dem. Det är viktigt att understryka att trots att det är teoretiskt möjligt att stjäla nycklar på detta vis, skulle det vara en oerhört svår process. Men hur svår den än är kan den inte förbises. När ett program skrivs som t.ex. hanterar privata nycklar, är det viktigt att göra en bedömning av de rådande omständigheterna och sedan besluta sig för om BigIntegers oföränderlighet utgör ett orosmoment.

För dem som verkligen är paranoida måste även operativsystemets paging mekanism beaktas (Viega & McGraw, 2002). Under tiden som ens program körs kan operativsystemet swapa ut delar av minnet till disk, inklusive känslig information som lösenord och privata nycklar. Det finns egentligen inget som kan göras åt detta i Java, förutom att rensa bort känslig information så fort som möjligt för att minska risken för att den swapas ut (Garms & Somerfield, 2001). Om ens applikation behöver denna nivå av säkerhet, är det nog dags att överlägga om det inte är nödvändigt att använda sig av inhemsk kod som kan låsa minnessegment och förhindra att de swapas ut.

Lagring och returnering av objekt och fält

När objekt och fält returneras är det viktigt att tänka på att det kan vara möjligt för det anropande objektet att modifiera det som returnerats (Sun Microsystems, 2000). Följande klass är ett bra exempel på detta (Garms & Somerfield, 2001):

```
public class Example
{
   private String[] internalData;


   public String[] getData()
   {
      return internalData;
   }
}
```

Det är möjligt att modifiera internalData med de följande raderna i någon annan kod, antagande att example är en instans av Example (Garms & Somerfield, 2001):

```
String[] array = example.getData();
array[ 0 ] = ”This is a new String.”;
```

I exemplet modifieras första elementet av det privata fältet internalData, som tillhör objektet example. Detta är troligen inte det ämnade beteendet. För att förhindra detta borde istället en kopia göras av fältet innan det returneras. På så sätt ändras inte det interna tillståndet av det ursprungliga objektet, även om det returnerade fältet modifieras.

Även när ett föränderligt objekt eller fält förs in i ett objekt för att användas som en intern variabel är det viktigt att göra en kopia (Garms & Somerfield, 2001). T.ex. följande klass:

```
public class Example
{
   private String[] internalData;

   public Example( String[] data )
   {
       internalData = data;
    }
}
```

Återigen finns det en möjlig sårbarhet om följande kod körs (Garms & Somerfield, 2001):

```
String[] someData = { ”String1”, ”String2” };
Example example = new Example( someData );
someData[ 0 ] = “New String”;
```

I det här skedet efter att det ovanstående exemplet körts, har internalData fältet blivit modifierat utan att gå genom example objektet. Lösningen i det här fallet är återigen att göra en kopia av fältet. Detta skall göras i konstruktorn som i följande exempel:

```
public Example ( String[] data )
{
   internalData = new String[ data.length ];
   System.arraycopy( data, 0, internalData, 0, data.length );
}
```

När någonting annat än ett fält lagras, är det möjligt att använda sig av clone() metoden (Garms & Somerfield, 2001). Om clone() metoden inte är tillgänglig, kan det vara nödvändigt att skapa ett nytt objekt och kopiera det ursprungliga objektets värden med get() och set() metoder.

Det går även att anropa clone() metoden på ett fält, men det är viktigt att förstå att varje objekt i fältet inte kommer att bli klonat, utan endast fältet på den högsta nivån (Garms & Somerfield, 2001). När det krävs kopiering på djupet, måste det utföras manuellt som i det ovanstående exemplet.

Vid lagring av oföränderliga (immutable) data typer, är det inte nödvändigt att oroa sig för att de skall bli modifierade i efterhand (Garms & Somerfield, 2001). T.ex. String kan inte

förändras när en instans väl skapats av den. Så det behöver inte göras någon kopia av den innan den returneras, även om den används som en intern variabel.

Serialisering

Serialisering möjliggör lagring och överföring av ett objekts tillstånd (Skansholm, 1999). Det används vanligtvis för att överföra objekt mellan virtuella maskiner och för att spara ett objekts tillstånd mellan en virtuell maskins instanser. När det kommer till serialisering, deserialisering och skapandet av objekt som kan serialiseras, är det viktigt att vara extra försiktig (Sun Microsystems, 2000). Objekt som är serialiserade står nämligen utanför Javas säkerhetssystem. Det är möjligt för någon som vanligtvis inte skulle kunnat modifiera ett objekt att modifiera det i dess serialiserade form.

Den förvalda inställningen vid serialisering är att alla interna variabler lagras till vilken angiven outputström som helst (Garms & Somerfield, 2001). Detta inkluderar även private deklarerade variabler, så det är viktigt att vara försiktig vid implementering av Serializable interfacet. Det är också viktigt att tänka på att subklasser kan implementera Serializable interfacet, vilket kan leda till att serialisering tillåts när det inte skall tillåtas. Det går att undvika detta fenomen genom att deklarera sina klasser som final eller genom att implementera de private deklarerade metoderna readObjetct(java.io.ObjectInputStream input), writeObject(java.io.ObjectOutputStream output) och se till att de kastar undantaget NotSerializableException som följande exempel:

```
private void readObject( ObjectInputStream input ) throws IOException, ClassNotFoundException
{
   throw new NotSerializableException(  “This class is not serializable”  );
}
private void writeObject( ObjectOutputStream input ) throwsIOException, ClassNotFoundException
{
   throw new NotSerializableException( “This class is not serializable” );
}
```

Detta förhindrar den virtuella maskinen att serialisera eller deserialisera en instans av den klass som koden finns i (Garms & Somerfield, 2001).

Transient

Transient nyckelordet används för att specificera variabler som inte är en del av ett objekts bestående tillstånd (Holzner, 2000). Om en variabel deklareras som transient, kommer den inte att serialiseras (Garms & Somerfield, 2001). Vid lagring av känslig data som inte skall exponeras för serialisering, är det möjligt att deklarera den som transient. Normalt deklareras inte en känslig klass som Serializable, fast det finns tillfällen då det kan vara nödvändigt.

En referens till en fil skulle tex. kunna deklareras som transient för att förhindra att en ogiltig referens används vid deserialisering i en annan virtuell maskin (Garms & Somerfield, 2001). Eftersom en variabel som deklareras som transient inte skrivs eller läses in vid anrop av metoderna writeObject(java.io.ObjectOutputStream output) och readObject(java.io.ObjectInputStream input), krävs det att den initialiseras på nytt vid tex. inläsning (Skansholm, 1999). På så sätt är det möjligt att förhindra att en ens applikation läser eller skriver till fel fil (Garms & Somerfield, 2001).

Validering

Det finns tillfällen då det är nödvändigt att validera fält i ett objekt när det deserialiseras (Garms & Somerfield, 2001). Det är en bra ide att validera fält som behöver vara internt konsekventa. För att uppnå detta är det möjligt att överskugga de två metoderna readObject() och writeObject().

Strömmarna anropar metoderna readObject() och writeObject() i ens klass vid serialisering och deserialisering av objekt av denna typ (Garms & Somerfield, 2001). Det är i dessa metoder som valideringen borde göras, som tex. att kontrollera att ett kontos balans stämmer eller att kontonumret är korrekt.

Kryptering

Ett annat sätt att skydda en byte ström utanför den virtuella maskinen är att kryptera strömmen som skapas av serialiseringspaketet (Sun Microsystems, 2000). Kryptering av byte strömmen förhindrar dekodning och läsande av ett serialiserat objekts privata tillstånd. Vid användande av kryptering måste man dessutom hantera nycklar, platsen där de lagras, sättet som de kommer att ges till deserialiseringsprogrammet, etc. Detta beror på att det inte finns något inbyggt stöd för nyckel hantering (Garms & Somerfield, 2001).

Paket

Grundinställningen när det kommer till paket är att ens Javakod kan vara i vilket paket som helst (Garms & Somerfield, 2001). Detta gör att vissa metoder och variabler kan bli utsatta för illasinnad manipulering. Eftersom en illasinnad klass kan ansluta sig till ett redan befintligt paket och på så sätt få tillgång till variabler och metoder med paket synlighet samt de som deklarerats som protected. Är det möjligt att med hjälp av tex. klassen IllasinnadOutPutStream som placeras i java.io paketet, att få tillgång till alla tidigare otillgängliga variabler och metoder i alla klasser i java.io.

Det finns tre sätt att skydda sig mot detta (Garms & Somerfield, 2001):

- Förseglade JAR filer
- Ett package.definition inlägg i java.security
- Begränsning av paketåtkomsten

Förseglade JAR filer

En förseglad JAR fil indikerar till den virtuella maskinen att alla klasser definierade i ett paket i JAR filen måste komma från just den JAR filen (Garms & Somerfield, 2001). Det innebär att ingen kan ansluta sig till ett paket i JAR filen från utsidan. För att försegla en JAR fil, krävs att det läggs till ett Name och Sealed inlägg i MANIFEST.MF filen för varje paket som skall förseglas. Om tex. com.isnetworks.* skall förseglas så att ingen kan definiera en klass i det paketet. Läggs följande rader till i MANIFEST.MF filen innan den signeras:

Name: com/isnetworks/

Sealed: true

Användande av package.definition

Det är även möjligt att lägga till ett package.definition inlägg för ett paket i java.security filen, vilket gör det omöjligt att skapa en klass i det paketet om inte vederbörliga rättigheter innehas (Sun Microsystems, 2000).

Om det skulle vara önskvärt att begränsa möjligheten att definiera klasser i tex. com.isnetworks.private.* och com.isnetworks.crypto.*, läggs helt enkelt följande rader till i java.security filen (Sun Microsystems, 2000):

Package.definition=com.isnetworks.private, com.isnetworks.crypro

Detta får klassladdarens defineClass metod att kasta ett undantag, när ett försök görs att definiera en ny klass inom dessa paket. Såvida inte koden beviljas följande tillstånd i java.policy filen (Sun Microsystems, 2000):

RuntimePermission(“defineClassInPackage.com.isnetworks.private”);

RuntimePermission(“defineClassInPackage.com.isnetworks.crypto”);

Begränsning av paketåtkomst

Även vilka klasser som har tillgång till ett visst paket kan begränsas (Sun Microsystems, 2000). Paket medlemmar kan skyddas från åtkomst av opålitlig kod genom att begränsa åtkomst till paketet och beviljande av åtkomst av endast specificerad kod. Detta kan åstadkommas genom att lägga till följande rader i java.security filen:

Package.access=com.isnetworks.*

Detta får klassladdarens loadClass metod att kasta ett undantag när ett försök görs att komma åt en klass i dessa paket, såvida koden inte beviljas följande tillstånd i java.policy filen (Sun Microsystems, 2000):

RuntimePermission(”accessClassInPackage.com.isnetworks”);

Privilegierad kod

Javaplattformens kontroll mekanismer skyddar systemresurser från ickeauktoriserad åtkomst, genom att försäkra sig om att den anropande koden har de vederbörliga rättigheter som krävs för att komma åt resurser (Sun Microsystems, 2000). Generellt när ett resursåtkomstförsök görs, måste all kod som traverseras av exekveringstråden fram till den punkten ha vederbörliga rättigheter för att åtkomsten skall tillåtas. Men det finns flera fall där system kod behöver komma åt en systemresurs för att kunna utföra sin funktionalitet, trots att den kod som anropade systemresurstjänsten inte har vederbörliga rättigheter för att komma åt resursen.

Ett exempel på ett sådant fall skulle kunna vara att klient kod har RuntimePermission att ladda ett visst inhemskt bibliotek (Sun Microsystems, 2000). Koden anropar loadLibrary tjänsten som för att kunna utföra sin uppgift, behöver läsrättigheter till den inhemska biblioteks filen. Men klient koden har inte de vederbörliga fil rättigheter som tillåter den att komma åt filen och om exekveringstråden skulle kollas för fil rättigheter, skulle operationen misslyckas när den egentligen skulle ha lyckats.

För att lösa detta problem har en API för privilegierad kod skapats, som möjliggör markerandet av ett kod block som privilegierat (Sun Microsystems, 2000). När ett kod block blir markerat som privilegierat, kan det anropa tjänster baserat på dess rättigheter även om vissa av dess anropare inte har dessa rättigheter.

Det är väldigt ovanligt att det är nödvändigt att använda sig av privilegierad kod (Garms & Somerfield, 2001). Oftast krävs det enbart för vissa systemnivåuppgifter, som redan har

vederbörliga rättigheter. Men om skrivande av privilegierad kod av någon anledning blir nödvändig, är det viktigt att följa dessa punkter:

- Vara försiktig vid överförande av parametrar
- Göra privilegierad kod så kort som möjligt
- Verkligen försöka att inte använda sig av privilegierad kod

Vara försiktig vid överförande av parametrar

Om ens privilegierade kod tar input parametrar, är det viktigt att vara medveten om att de inte alltid är det som var ämnat (Garms & Somerfield, 2001). Tex. kod som vill läsa systemegenskapen java.version skulle kunna implementeras på följande felaktiga sätt:

```
public String getProperty( final String property )
{
   return ( String ) AccessController.doPrivileged( new PrivilegedAction() {
      public Object run()
      {
         return System.getProperty( property );
      }
   } );
}
```

Problemet med ovanstående kod är att parametern som förs in i metoden inte behöver vara java.version, utan skulle kunna vara något som tex. username (Garms & Somerfield, 2001). Vilket gör det möjligt för någon att tex. få reda på en användares login namn. Det vore mycket lämpligare att skriva metoden på följande sätt:

```
public String getJavaVersion()
{
   return ( String ) AccessController.doPrivileged( new PrivilegedAction() {
      public Object run()
      {
         return System.getProperty( “java.version” );
      }
   } );
}
```

Genom att ta bort parametern som överförs till det priviligierade blocket, tas även möjligheten att använda sig av det på ett olämpligt sätt bort (Garms & Somerfield, 2001).

Göra privilegierad kod så kort som möjligt

Genom att ha så kort privilegierad kod som möjligt, reduceras antalet möjliga attack vägar (Garms & Somerfield, 2001). Vilket gör det lättare att försäkra sig om att ens kod inte kan exploateras.

Verkligen försöka att inte använda sig av privilegierad kod

Privilegierad kod skall användas sparsamt om alls (Sun Microsystems, 2000). Kod skall skrivas utan användande av privilegierade block, endast när ens kod stöter på säkerhetsundantag är det lämpligt att överväga att använda sig av privilegierade block.

Inhemska metoder

Möjligheten att anropa en inhemsk metod inifrån Java begränsas av Javas säkerhetshanterare (Garms & Somerfield, 2001). Men när väl en inhemsk metod är under exekvering, står den utanför säkerhetshanterarens kontroll och kan göra allt som tillåts av det underliggande operativsystemet. Därför är det viktigt att begränsa åtkomsten till potentiellt farlig kod, samt vad som kan föras in som argument till inhemska metoder och vad som kan returneras.

Det är viktigt att kontrollera inhemska metoder efter (Sun Microsystems, 2000):

- Vad de returnerar
- Vad de tar som parametrar
- Om de kringgår säkerhetskontroller
- Om de är public, private,….
- Om de innehåller metodanrop som kringgår paketgränser, dvs. kringgår paketskydd

En annan aspekt som är viktig att ha i åtanke när det kommer till inhemska metoder, är att de kan komma åt och modifiera objekt utan att deras aktivitet kontrolleras av Javas säkerhetshanterare (Garms & Somerfield, 2001). Detta innebär att om tex. en String förs in i en inhemsk metod, kan metoden modifiera objektet trotts faktumet att alla String objekt är oföränderliga (immutable) i den virtuella maskinen. Förutom oföränderliga objekt kan även private deklarerade medlemsvariabler och metoder som normalt vore oåtkomliga kommas åt av inhemsk kod om de förs in i en inhemsk metod.

Diskussion

”We wouldn’t have to spend so much time, money, and effort on network security if we didn’t have such bad software security.”, skriver Viega och McGraw (2002) i sin bok “Building Secure Software”. Detta återspeglar tydligt den rådande säkerhetsmässiga värkligheten inom IT idag. Modifierade paket som tar ner servrar, milliontals olika buffer-overflow attacker och olika krypterings sårbarheter. Alla dessa är för det mesta mjukvarubaserade problem, som måste eller åtminstone borde konfronteras på kod nivå. Ibland är det visserligen möjligt att skydda sig mot dessa sårbarheter med hjälp av olika verktyg och hjälpmedel som t.ex. brandväggar. Men eftersom det grundläggande problemet ligger i själva mjukvaran, är det även där som själva åtgärderna bör koncentreras för att uppnå bästa möjliga resultat.

När det kommer till Java är det dessutom oerhört viktigt att inse att det inte finns något substitut för bra designad kod. Trots det faktum att Java har en hel del inbyggda säkerhetsmekanismer som underlättar det för programmerare säkerhetsmässigt, har det visat sig att dessa ofta fallit offer för design problem. Därför får inte betydelsen av kodens design underskattas.

Genom att uppdaga olika möjliga attack vägar och programmeringstekniker är det möjligt att förbättra kod designen och minska antalet frekvent upprepade misstag, som leder till diverse säkerhetshål vid skrivande av Javakod. Seriösa programmerare har inte råd att misslyckas inom detta område. En applikation som har allvarliga grundläggande säkerhetsbrister pga.

dåligt skriven kod är inte värd mycket idag. Oavsett om dess funktionalitet i övrigt är tillfredställande.

Många av de möjliga attack vägar som behandlas i uppsatsen, är det troligen osannolikt att stöta på i det verkliga livet. Självklart beror det på skalan av ens applikation. Men trots detta är det oerhört viktigt att vara medveten om de konsekvenser som Javakod kan innebära säkerhetsmässigt.

För att idag kunna kalla sig själv för en seriös programmerare är det dessutom nödvändigt att hålla sig uppdaterad inom området och följa riktlinjer samt använda sig av den ofantliga erfarenhet som finns ute bland andra ofta mer erfarna utvecklare. På så sätt är det möjligt att lära sig att känna igen och inte upprepa misstag som gjorts av andra gång på gång. För att poängtera detta skriver bl.a Gonsalves (2002): "Most of the time -- well over 90 percent of the time -- vulnerabilities are equivalent to somebody forgetting to nail down the shingle that blew off the roof, Hernan said. We tend to see the same kinds of mistakes being made over and over again. We see that in open-source software and in closed-source software."

Slutsatser

Studien av den aktuella problemformuleringen har lett till slutsatsen, att Javakod säkerheten ökar avsevärt om de följande fem punkterna täcks säkerhetsmässigt:

- Åtkomst
- Serialisering
- Paket
- Privilegierad kod
- Inhemska metoder

Dessa områden bedöms som extra känsliga när det kommer till säker Javakod och blir ofta förbisedda av orutinerade programmerare. Genom att se till att inte upprepa tidigare misstag och följa riktlinjer för olika programmeringstekniker inom dessa fem områden, är det möjligt att skapa en säkrare applikationsgrund. Självklart måste även en hel del andra aspekter beaktas beroende på vidden av ens applikation. Men genom att beakta de fem ovanstående områdena, kan en gedigen grund uppnås som är säkrare att bygga vidare på.

Referenser

Böcker

Garms, Jess., & Somerfield, Daniel. (2001). Java Professional Security. Wrox Press.

Oaks, Scott. (2001). Java Security, 2nd Edition. O'Reilly.

Howard, Michael., & LeBlanc, David. (2002). Writing Secure Code, Second Edition. Microsoft Press.

Viega, John., & McGraw Gary. (2002). Building Secure Software. Addison-Wesley.

Skansholm, Jan. (1999). Java Direkt. Studentlitteratur.

Gong, Li. (1999). Inside Java 2 Platform Security. Addison-Wesley.

Bakharia, Aneesha. (2001). Java Server Pages. Prima Publishing.

Clements, Allan. (2002). The Principles of Computer Hardware. Oxford University Press.

Harold, Rusty, Elliot. (1997). Hemligheterna i Java. IDG AB.

Holme, Magne, Idar., & Solvang, Krohn, Bernt. (1997). Forskningsmetodik. Studentlitteratur.

Hellevik, O. (1980). Forskningsmetode i sosiologi og statsvitenskap. Holt & Winston.

Holzner, Steven. (2000). Java Black Book. The Coriolis Group.

Internet

Somayaji, Nanjunda. (2002). Implementing Java applications security. [online document 2002-11-12]. URL http://www.serverworldmagazine.com/monthly/2002/05/java.shtml

CSI/FBI. (2002). Computer Crime and Security Survey. [online document 2002-12-13]. URL http://www.gocsi.com/press/20020407.html

Lange, Larry. (1997). The Rise of the Underground Engineer. [online document 2002-12-10]. URL http://www.blackhat.com/media/bh-usa-97/blackhat-eetimes.html

Sundsted, Todd. (2001). Secure your Java apps from end to end, Part 1. [online document 2002-11-20]. URL http://www.javaworld.com/javaworld/jw-06-2001/jw-0615-howto.html

Sundsted, Todd. (2001). Secure your Java apps from end to end, Part 2. [online document 2002-11-20]. URL http://www.javaworld.com/javaworld/jw-06-2001/jw-0615-howto.html

Sundsted, Todd. (2001). Secure your Java apps from end to end, Part 3. [online document 2002-11-20]. URL http://www.javaworld.com/javaworld/jw-06-2001/jw-0615-howto.html

Sun Microsystems. (2000). Security Code Guidelines.[online document 2002-12-16]. URL http://java.sun.com/security/seccodeguide.html#gcg

McGraw, Gary., & Felten, Edward. (1998). Twelve rules for developing more secure Java code. [online document 2002-12-16]. URL http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html

Mogasale, Sudarshan N. (2000). Access Violations [online document 2002-12-05]. URL http://sunsite.uakom.sk/javafaq/reports/accessviolations.html

Gonsalves, Antone. (2002). Is Open-Source Software Less Secure? [online document 2002-11-25]. URL http://www.techweb.com/wire/story/TOE20021121S0001