Utvecklande av en mer flexibel och användbar Security Manager i Java

Linus Engback och Fredrik Åslin
Dept. of Computer and Information Science, Linköpings universitet
{linen351, freas561 }@student.liu.se

Abstract

Denna rapport handlar om implementerandet av en användarvänlig security manager för Java. Författarna har implementerat en förlängning av Suns Java security manager som promptar användaren då systemets policy utmanas och tillåter användaren att besluta om handlingen skall tillåtas. Denna rapport ger en allmän beskrivning av arbetet samt detaljer om hur implementationen har gjorts.

1. Introduktion

Programmeringspråket Java från Sun Microsystems tillhandahåller en lösning för att skydda den ”sandlådemiljö” som Java använder sig av. Värt att poängtera är att denna ej används som standard utan explicit måste startas.

Javas lösning bygger runt en security manager som fångar upp alla potentiellt farliga anrop och kontrollerar om koden har tillåtelse att göra detta anrop. Normalt specificeras vilken tillgång en given applikation har i en policyfil. Security managern fungerar genom att den läser de angivna policyfilerna och letar efter de permissions som behövs för att tillåta anropet. Om den inte finner passande permissions så kastas ett undantag och via detta nekas koden tillgång.

För att kunna bestäma vilka permissions som ett program har så introducerade Java skyddsdomäner som kapslar in kod och dess permissions. Med det menas att varje klass kommer att tillhöra en given domän som avgör vad klassen får eller inte får göra. När security managern undersöker ett givet anrop så traverserar den anropskedjan och kontrollerar samtligas skyddsdomäner. Vid traverseringen kontrolleras huruvida huruvida de tillåter anropet eller inte. Alla skyddsdomäner måste tillåta anropet för att det ska tillåtas, annars kommer ett undantag att kastas.

2. Kravspecifikation

Projektets kravspecifikation kan hittas på http://www.ida.liu.se/~TDDC03/projects/prjlist.shtml. I korta drag specificeras att en security manager ska tas fram som befriar användaren från att behöva redigera en aktuell policyfil. I stället ska användaren bara behöver ta ett beslut då ett program försöker göra något som ej är tillåtet enligt den gällande policyn. Användaren ska då ställas inför valen ”Tillåt nu”, ”Tillåt för evigt” och ”Tillåt inte”. Det första valet innebär att programmet tillåts göra det som policyn själv ej tillät. Det andra valet att policyfilen ändras till att tillåta det som skulle göras och tredje valet att programmet returnerar det ursprungliga undantaget som kastades.

3. Implementation

En av de viktigare klasserna som implementerar säkerheten hos Java är java.security.SecurityManager, som hanterar alla säkerhetskontroller. Den är byggd så att den har ett antal metoder för att kolla permissions, men alla de metoderna i sin tur anropar SecurityManager.checkPermission() för att göra själva säkerhetschecken. Denna implementation öppnar för att göra en subklass till SecurityManager och överlagra denna metod för att kunna fånga upp de undantag som genereras. Detta räcker eftersom de enda gånger något behöver göras är när något nekas tillgång, vilket innebär att vad som nekas samt i vilken miljö den nekas i kommer att fångas.

Genom att fånga upp undantaget så implementeras enkelt de val användaren skulle ges. Det första valet skulle ge användaren möjligheten att tillåta anropet en gång. Detta görs genom att fånga upp undantaget utan åtgärd. I detta fall kommer samma undantag att genereras även vid nästa exekvering. Användaren skulle även ges möjlighet att neka anropet tillgång, vilket implementeras genom kasta samma undantag igen. Detta kommer att traversera anropskedjan och om det ej fångas kommer det att kastas från toppnivån. Sista valet var att alltid tillåta anropet, vilket implementerades genom att undantaget fångas upp, det som behövs läggs till i policyfilen och till sist återgår kontrollen till den metod som genererade undantaget.

Som det nämndes ovan så måste varje skyddsdomän

tillåta anropet för att anropet i sin helhet ska tillåtas. Dock finns det ett undantag till detta. Det är när någon kod körs som priviligerad kod. Priviligerad kod är kod är märkt som pålitlig, vilket ger den större tillgång än vad annars vore fallet. När en kod körs som priviligerad så behöver den bara permission för att göra de anrop som den själv gör, och behöver då inte ta hänsyn till vilken skyddsdomän den anropas ifrån. Detta implementeras via java.security.AccessControlContext som kapslar in den miljö som anropen sker i samt dess skyddsdomäner. När en kod körs som privilegerad kommer en ny AccessControlContext att skapas där den priviligerade kodens skyddsdomäner och den icke priviligerade delens skyddsdomäner skiljs åt. Därefter använder den sig av en java.security.DomainCombiner för att kombinera de två delarna när det behövs.

I Java är både AccessControlContext och AccessController slutgiltiga klasser. Detta innebär att det inte kan ärvas från dessa klasser. Det enda sätt vi funnit att nå de skyddsdomäner som ligger under ”run time”-miljön är att implementera klassen DomainCombiner. I denna implementerar vi den abstrakta metoden combine vilket gör det möjligt att nå de skyddsdomäner som är lagrade i miljön.

Genom detta kan de skyddsdomäner som är markerade som priviligerade nås och vilka permission som saknas undersöks. I fallet att ingen kod körs som priviligerad så fångas anropskedjan och varje del i den kontrolleras. Oavsett fall så kommer ett set av skyddsdomäner som inte tillåter anropet att hittas.

I enlighet med kravspecifikation använde vi oss av Suns policy parser klass. Denna tillhandahåller metoder för att skriva, läsa och traversera policyfiler. Eftersom ett undantag har genererats vet vi att någon av de samlade domänerna inte har den permission som krävs. Därför behöver det ej kontrolleras om den felande domänen redan har denna permission i policyfilen. I policyfilen använder man sig av var koden kommer ifrån för att skilja dem åt. Denna information lagras som skyddsdomänens kodbas och är en sökväg.

Den mest rättframa lösningen för att lägga de permissions som behövs i policyfilen vore att addera dem i slutet av filen. Dock skulle detta ge en dålig struktur i filen. Därför traverseras policyfilen istället för att finna en eventuell kodbas som matchar skyddsdomänens. Om en sådan hittas så läggs den aktuella permissionen till, annars skapas en ny kodbas. Det är sannolikt att detta tar något längre tid än att lägga till det i slutet av filen men detta uppvägs av bättre den strukturen.

4. Resultatet

Vårt demonstrationsprogram består av tre klasser där klass A anropar klass B och klass B anropar klass C. Varje klass utför en filläsning. Klass B utför all sin kod som priviligerad kod. Detta innebär klass A inte kommer att behöva permissions för att kunna läsa de filer som läses i B och C. Däremot behöver klass B permission för att läsa från fil b och c.

Policyfilen innehåller från början nedanstående permissions.

```
/* AUTOMATICALLY GENERATED ON
   Mon May 08 15:38:14 CEST 2006*/
/* DO NOT EDIT */

grant codeBase
"file:/home/freas561/pue/pack/B.jar" {
  permission java.io.FilePermission
  "/home/freas561/b.txt", "read";
};

grant codeBase
"file:/home/freas561/pue/pack/C.jar" {
  permission java.io.FilePermission
  "/home/freas561/c.txt", "read";
};
```

Visa flaggor måste sättas för att rätt security manager ska användas med rätt permissions. Flaggan -Xboothclasspath används för att den implementerade security managern ska köras som systemkod. Denna är nödvändig är för att permissions ej ska behöva ges till den egna security managern. Flaggan -Djava.security.manager specificerar en egen security manager som ska användas. Flaggan -Djava.security.policy ger användaren möjlighet att specifiera vilken policyfil som ska användas.

```
java -Xbootclasspath/a:pack/tddc03.jar
-cp .:pack/A.jar:pack/C.jar:pack/B.jar
-Djava.security.manager=tddc03.SecMan2
-Djava.security.policy=java.policy
A.openFileWithPrivA

Securityexception caught:
access denied
(java.io.FilePermission
  /home/freas561/a.txt read)
Would you like to:
1. Allow once,
2. Allow everytime,
3. Deny?
% 1
Line from A:This if file a.
Calling openFileBPrivleged()
Line from B:This is file b.
Calling openFileC()
Securityexception caught:
access denied
(java.io.FilePermission
  /home/freas561/c.txt read)
Would you like to:
```

```
1. Allow once,
2. Allow everytime,
3. Deny?
%2
Line from C:This is file c.
```

I körningen genereras två undantag. Det första genereras på grund av att klass A saknar den permission som krävs för att läsa fil a. När detta sker väljer användaren att tillåta klass A att läsa den associerade filen endast denna gång. Ingen uppdatering av den angivna policyfilen sker.

I nästa steg generas ett undantag när klass C försöker läsa fil C. Detta undantag generaras på grund av att inte bara klass C utan även de metoder som anropar klass C måste ha permissions att läsa. I detta fall är anropskedjan klass A, klass B, klass C vilket normalt skulle innebära att samtliga tre klasser måste ha permission att läsa fil c. Men i detta fall görs anropet till klass C från priviligerad kod i klass B, vilket avbryter kedjan vid klass B. Klass A måste alltså inte ha permission att läsa fil c. Konsekvensen blir att användaren måste ta ställning till om klass B ska få läsa fil c som läses i klass C. I körexemplet tillåter användaren detta och vill att det även ska tillåtas i framtiden. Detta sparas som en ändring i den angivna policyfilen.

Resultatet av körningen blir att endast klass B:s permissions i den angivna policyfilen kommer att uppdateras. Nedan ses policyfilen efter körning där kodbasen för klass B har uppdaterats med läsrättigheter till fil c.

```
/* AUTOMATICALLY GENERATED ON
   Mon May 08 15:39:13 CEST 2006*/
/* DO NOT EDIT */

grant codeBase
"file:/home/freas561/pue/pack/B.jar" {
  permission java.io.FilePermission
  "/home/freas561/b.txt", "read";

  permission java.io.FilePermission
  "/home/freas561/c.txt", "read";
};

grant codeBase
"file:/home/freas561/pue/pack/C.jar" {
  permission java.io.FilePermission
  "/home/freas561/c.txt", "read";
};
```

5. Introducerade säkerhetsrisker

I ett system med en aktiv security manager avgör policyfilen vilka program eller vem som får göra vad. Om någon entitet olovligen kan ändra policyfilen ger detta den en möjlighet att bestämma vad den själv ska få utföra. Detta innebär att policyfilen i sig själv är en säkerhetsrisk. Därför är det viktigt att operativsystemet skyddar policyfilen. I detta sammanhang är det också viktigt att tänka på att security managern kan ge en falsk känsla av säkerhet. I ett system där man vanligtvis inte begränsar rättigheter måste policyfilens aktivt skyddas.

Det har varit viktigt för oss att inte riskera att bygga in extra säkerhetshål eftersom vi har arbetat med lösningar för säkerhet. I slutändan efter konsultation med Almut Herzog beslöt vi oss för att ändå låta hela vår security manager köras som systemkod. Detta eftersom det annars är lätt att få slingor där vår security manager kräver permissions, vilket är vad den själv är avsedd att hantera.

6. Framtida arbete

Vår security manager fungerar och skulle kunna användas redan idag av personer med rätt teknisk bakgrund. Dock skulle en del vidareutveckling krävas för att den ska bli bra nog att användas av vanliga konsumenter. Gränssnittet måste bli bättre, framförallt måste problemen med dialogen gentemot användaren lösas. Information i mer lättförståelig form om vad olika undantag innebär vore önskvärt. Vår security manager skulle också behöva kombineras med mjukvara för redigering av policyfilen. I nuläget kan man lägga till permissions, men det saknas funktioner för att ta bort dom eller på att annat sätt för hand redigera dom. En installerare av samma typ som finns i Windowssystem vore också önskvärt.
Att implementera en sådan här lösning som gör användandet enklare har nackdelen att den tillåter användarna att ta ogenomtänkta beslut. Det är inte orimligt att ett sådant system vi implementerat uppmuntrar ett ”click-happy”-beteende. Av denna anledning skulle det vara önskvärt att systemet ger tydlig information om varje val och att antalet val hålls nere. Detta ligger dock utanför omfattningen för detta projekt.

7. Sammanfattning och slutsats

Vårt arbete med detta projekt har haft två faser: informationssökning och implementation. Informationssökningen har varit den del av arbetet som tagit mest tid. Vi har studerat delar av Javas API ingående, läst kod, sökt efter liknande projekt och testkört under mer än 2/3 av projekttiden.

Vi har kunnat konstatera att mycket få liknande projekt finns implementerade. En anledning vi har kunnat tänka oss är om denna typ av lösning har någon egentlig marknad. Det är oftast den som är ansvarig för systemet som har kunskaperna att avgöra vad som ska tillåtas, och då bör de gängse metoderna att hantera policys vara nog.