Säker datahantering med SE-PostgreSQL

A N D R E A S   C E D E R H O L M
o c h   M A T T I A S   L I N D S T R Ö M

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2011

Säker datahantering med SE-PostgreSQL

A N D R E A S   C E D E R H O L M
o c h   M A T T I A S   L I N D S T R Ö M

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2011
Handledare på CSC var Mads Dam
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/
cederholm_andreas_OCH_lindstrom_mattias_K11010.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Referat

SE-PostgreSQL är ett tillägg till databashanteraren PostgreSQL som samordnar med SELinux för att lägga till obligatorisk åtkomstkontroll i databaser. Den obligatoriska åtkomstkontrollen går dessutom att applicera på objekt på lägre nivå än vad som är möjligt med den vanliga åtkomstkontrollen i PostgreSQL, närmare bestämt även på rader och kolumner.

Rapporten börjar med att beskriva hur systemen SELinux, PostgreSQL och SE-PostgreSQL fungerar tillsammans. Rapporten fortsätter sedan med att presentera ett antal tester som utfördes för att ta reda på hur väl åtkomstkontrollen på rad- och kolumnnivå fungerar samt hur Multi-Category Security kan användas i en databas.

Abstract

SE-PostgreSQL is an extension to the database management system PostgreSQL which used together with SELinux adds mandatory access control to databases. Unlike the regular access control in PostgreSQL, it is possible to apply the mandatory access control to low level objects like rows and columns.

This essay begins with describing how the systems SELinux, PostgreSQL and SE-PostgreSQL works together. The essay then proceeds with presenting a number of tests performed to determine how one can use Multi-Category Security in a database and whether the access control on rows and columns gives a satisfactory result.

Innehåll

1 Introduktion **1**
1.1 Bakgrund 1
1.2 Hantering av åtkomst 1
1.2.1 DAC och MAC 1
1.2.2 SELinux 2
1.2.3 PostgreSQL 5
1.2.4 SE-PostgreSQL 5
1.3 Syfte 6

2 Test **7**
2.1 Databasen 7
2.2 Tester 8
2.2.1 INSERT 8
2.2.2 UPDATE 9
2.2.3 SELECT 10
2.2.4 Aggregerande funktioner 12
2.2.5 Vyer 14
2.2.6 Användning av “\d” 15

3 Diskussion **19**
3.1 Säkerhet 19
3.2 Användarvänlighet 20
3.3 Design av databas 21
3.3.1 Normalform och Säkerhet 21
3.3.2 Val av primärnycklar 23

4 Slutsatser **25**

Litteraturförteckning **27**

Bilagor **28**

A Testsystemet **29**

Kapitel 1

Introduktion

1.1 Bakgrund

När allt större mängder data görs tillgänglig på Internet och i privata nätverk blir det viktigt att kontrollera att ingen information blir tillgänglig för någon annan än behöriga användare. I moderna databashanterare finns möjligheter att dela ut behörighet till databaser, tabeller, vyer, mm. Något som dock ofta saknas är möjligheten att tilldela behörighet för specifika tupler eller kolumner. Security-Enhanced PostgreSQL, SE-PostgreSQL, är ett tillägg till databashanteraren PostgreSQL som gör det möjligt att tilldela dessa mer specifika behörigheter. Utöver detta är målet med SE-PostgreSQL att rättigheterna i databaserna inte ska vara självständiga från rättigheterna i filsystemet. Rättigheterna till filsystemet hanteras vanligtvis av operativsystemet och tanken är att operativsystemet även ska ha samma övergripande kontroll över databaserna. SE-PostgreSQL fungerar tillsammans med Security-Enhanced Linux, SELinux, vilket är en säkerhetsmodul till operativsystemet Linux [1].

1.2 Hantering av åtkomst

Det här avsnittet förklarar hur SE-PostgreSQL hanterar åtkomsträttigheter i databaser. Det börjar med att generellt beskriva hur DAC och MAC fungerar och deras respektive för- och nackdelar. Avsnittet fortsätter sedan med att förklara hur SELinux fungerar och hur SE-PostgreSQL skiljer sig från vanliga PostgreSQL inklusive dess koppling till SELinux.

1.2.1 DAC och MAC

Discretionary Access Control, DAC, är den traditionella metoden för att hantera åtkomsträttigheter. Metoden går ut på att varje enskild användare eller roll är ett subjekt som kan tilldelas rättigheter såsom att till exempel läsa eller skriva till ett objekt i systemet. Vanligtvis blir den användare som skapar ett objekt ägare av

objektet och får utöver fulla rättigheter att använda objektet även rätten att föra vidare rättigheter till andra användare. Dessutom är det möjligt för alla användare att föra vidare de rättigheter som man har fått sig tilldelade[2].

Det finns ett antal nackdelar med åtkomstmodellen DAC. För det första förutsätter modellen att man kan lita på att ägarna av objekten är helt pålitliga men framför allt avspeglar metoden dåligt hur en dator fungerar eftersom rättigheterna är kopplade till användare. När man ger en användare tillåtelse att ändra på innehållet i en fil har man egentligen gett rättigheten till alla program som användaren kör. Detta trots att användaren sällan vet exakt vad ett program gör. Det här skulle inte vara något problem om vi kunde lita på att alla program gör vad de ska och absolut ingenting annat. Det kommer dock alltid att finnas kod med säkerhetsbrister och kod som är medvetet skriven för att göra något annat än vad användaren tror, det vill säga trojaner[9].

Med Mandatory Access Control, MAC, menas att man överlåter kontrollen av åtkomsträttigheter på systemet utan inblandning från användare, ägare eller administratörer. Vanligtvis implementeras någon form av policy vilken alla subjekt måste följa och som systemet kontrollerar inför varje åtkomstförsök till ett objekt. Med en obligatorisk säkerhetspolicy är det möjligt att minimera skadan som kan uppstå på grund av trojaner. MAC är i sig självt inte en fullständig säkerhetslösning eftersom flexibilitet i exempelvis utbyte av information inom ett företag är något som behövs samtidigt som avsaknaden av flexibilitet är en stor del av den säkerhet som MAC erbjuder.

I system som implementerar MAC brukar därför även finnas en implementation av DAC. Den obligatoriska policyn används vanligtvis för att definiera övergripande domäner inom vilka ett subjekt får ha speciella rättigheter till olika objekt medan diskreta rättigheter används för att specificera mer detaljerade rättigheter för enskilda subjekt. Det betyder exempelvis att man endast diskret skulle kunna ge ett subjekt läsrättigheter till ett specifikt objekt om subjektets domän har tillåtelse att läsa filer av angivet objekts typ. Den mest välkända modellen som bygger på MAC är Multi-Level Security, MLS. MLS tillgodoser dock bara ett begränsat säkerhetsbehov, närmare bestämt att information inte ska kunna flyta mellan olika sekretessnivåer.[3][10][11].

1.2.2 SELinux

Security-Enhanced Linux, SELinux, är en säkerhetsmodul till operativsystemet Linux som implementerar en mer generell version av MAC utöver den diskreta åtkomstkontroll som redan finns.

Den teoretiska utvecklingen av den arkitektur som utgör SELinux påbörjades på nittiotalet av National Security Agency, NSA, och Secure Computing Corporation. Tillsammans med en forskningsgrupp från University of Utah utvecklades sedan en prototyp till forskningsoperativsystemet Fluke, arkitekturen fick namnet Flask. För att ge tekniken en större användarbas valde NSA att överföra Flask-arkitekturen till operativsystemet Linux. I december år 2000 släpptes den överförda arkitekturen

som öppen källkod under namnet Security-Enhanced Linux, SELinux [4][5].

Linux använder sig av en version av DAC för att reglera åtkomst till exempelvis filer. Den som skapar en fil kan sätta rättigheter för att andra användare ska kunna läsa, skriva respektive köra filen. SELinux implementerar ett extra lager åtkomstkontroll utöver det redan existerande. Den extra åtkomstkontrollen är i form av MAC. Användning av SELinux betyder således att man använder ett system som implementerar både DAC (Linux) och MAC (SELinux).

Subjekten i Linux åtkomstkontroller (DAC) är användarna. I SELinux separeras användarna från de processer de skapar genom att varje enskild process behandlas som ett eget subjekt. Alla användare, subjekt och objekt har sedan tilldelats en så kallad Security Context som tillsammans med en samling regler, Security Policy, används för att avgöra huruvida åtkomst är tillåten och på vilket sätt. För att ingen process ska göra något oönskat är allt som inte explicit tillåts i reglerna förbjudet [16].

En Security Context består av fyra huvuddelar på formen:

```
användare:roll:typ:mls/mcs
```

Den del som framför allt används för att reglera åtkomst är typ, av historiska skäl även kallad för domän när man talar om subjekt. Regelsamlingen består främst av tillåtelser till operationer mellan subjekt och objekt identifierade med olika typer. Reglerna beskriver på vilket sätt subjekt av en bestämd typ har tillåtelse att påverka objekt av en annan bestämd typ. Om reglerna inte innehåller ett samband mellan två typer tillåts inte subjektet påverka objektet. Detta brukar kallas för Type Enforcement.

Användaren i en SELinux Security Context ska inte förväxlas med den vanliga Linux-användaren, men de är besläktade. Alla Linux-användare i systemet kopplas till en SELinux-användare som används för att identifiera vilka roller Linux-användarens processer kan anta och maximala gränserna för det MLS/MCS-intervall användaren har tillgång till (en enskild Linux-användare kan begränsas ytterligare i storleken på intervallet). När Linux-användaren startar en process följer därför SELinux-användaren med i processens egna Security Context. Om processen sedan skapar ett nytt objekt eller subjekt förs SELinux-användaren vidare. Även processer och objekt som inte är skapade av en Linux-användare får en SELinux-användare (oftast system_u).

Rollen i en Security Context finns med för att tillåta rollbaserad åtkomstkontroll i SELinux. Rollbaserad åtkomst i SELinux är inte samma sak som rollbaserad åtkomst i Linux vanliga åtkomstkontroll. SELinux rollbaserade åtkomst sker genom att en roll definierar vilka typer som kan antas, eftersom rättigheter till åtkomst utgår från typen. Hierarkin användare, roll och typ definierar tillsammans vilka typer en Linux-användares processer kan anta och sätter därmed en begränsning på vad olika användare får göra i systemet. Rollen är dock inte tillräckligt för att tillåta ett byte mellan två typer. Varje möjligt byte mellan två givna typer måste dessutom definieras explicit när och hur det ska ske i reglerna. Eftersom objekt aldrig har

behov av att byta typ är roll endast relevant för subjekt. Alla objekt blir därför bara tilldelade ett standardvärde (oftast object_r) [6][12] [13].

Den sista delen finns med för att tillåta Multi-Level Security, MLS, enligt Bell-LaPadula-modellen och även Multi-Category Security, MCS. Det är möjligt att ange en eller flera sekretessnivåer, sensitivities, och i enighet med Bell-LaPadula-modellen kan man inte skriva till en sekretessnivå som är lägre än den egna eller läsa från en som är högre. MCS handlar om att man dessutom kan tilldela så kallade kategorier till både subjekt och objekt. Endast objekt vars kategorier är en delmängd av de kategorier subjektet har tillgång till får läsas. Den tydligaste skillnaden mellan MLS och MCS är således att det inte finns någon intern ranking mellan kategorierna i MCS och därmed inte heller några begrepp som högre och lägre kategorier. Man kan aldrig vare sig läsa eller skriva utanför de kategorier man har tillgång till. För att få tillåtelse att läsa ett objekt som utnyttjar den sista delen av en Security Context måste alltså objektets sekretessnivå vara lägre eller eller lika med subjektets sekretessnivå samtidigt som objektets kategorier är en delmängd av subjektets tillåtna kategorier [14] [15].

Vilka kategorier, och sekretessnivåer, en användare har rätt att använda går att ändra med hjälp av verktyget semanage. En Linux-användare kan aldrig ha tillgång till kategorier, och sekretessnivåer, som inte är tillgängliga för den SELinux-användare kontot är kopplat till [19]. Man kan lista alla SELinux-användare som finns på systemet med information om möjliga roller och MLS/MCS med kommandot:

```
# semanage user -l
```

Det är sedan möjligt att ändra MLS/MCS-intervall, det vill säga mängderna av tillåtna sekretessnivåer och/eller kategorier, för enskilda SELinux-användare med kommandot:

```
# semanage user -m -r <nytt intervall> <vald SELinux-användare>
```

Det är möjligt att skapa nya SELinux-användare men det är oftast onödigt och kräver att man skriver nya regler till policyn. När man lägger till en användare i systemet kan man direkt koppla den till en existerande SELinux-användare på följande vis:

```
# useradd -m -Z <SELinux-användare> <användarnamn>
```

Man kan sedan ändra på MLS/MCS-intervallet för varje enskild användare inom gränserna för vad SELinux-användaren tillåter med hjälp av semanage:

```
# semanage login -m -s <SELinux-användare> -r <intervall> <användare>
```

1.2.3 PostgreSQL

Stora mängder konfidentiell data sparas vanligtvis i databaser och det är därför viktigt att säkerhetsfunktionerna som erbjuds av databashanteringssystemen hindrar att data manipuleras på sådant sätt att det görs tillgängligt för en obehörig användare.

PostgreSQL är ett vanligt databashanteringssystem med öppen källkod som började utvecklas 1986 av Michael Stonebraker vid University of California, Berkley, då under namnet Postgres. År 1996 togs projektet upp av utvecklare utanför den akademiska världen och döptes om till PostgreSQL [7].

Åtkomst till data i PostgreSQL hanteras av en version av DAC. Hur åtkomst till ett objekt i PostgreSQL ska gå till bestäms av ägaren till objektet i fråga. Den som skapar ett objekt, såsom en databas eller en tabell, får alla rättigheter till det. Det inkluderar, förutom att exempelvis titta på informationen eller lägga till något nytt i objektet, rätten att tilldela andra användare rättigheter till objektet. Fortsättningsvis är det möjligt att, när man tilldelar rättigheter till andra användare, dessutom ge dem tillåtelse att föra de givna rättigheterna vidare. Vid sidan av rättighetshanteringen står superanvändarna. Superanvändarna fungerar som om de vore ägare till alla objekt och har därmed samma rättigheter till varje objekt som dess ägare har [8].

1.2.4 SE-PostgreSQL

SE-PostgreSQL är ett tillägg till databashanteraren PostgreSQL för användning tillsammans med SELinux. Tillägget utökar PostgreSQLs system av diskreta rättigheter med att skapa åtkomstregler baserade på säkerhetspolicyn i SELinux. Alla objekt som hanteras i databashanteraren tilldelas precis som i SELinux filsystem en Security Context, i SE-PostgreSQL kallas det för Security Label. Det första steget vid bedömning om en fråga till databasen ska vara tillåten utförs av PostgreSQLs vanliga diskreta säkerhetsmekanism beskriven ovan och om åtkomst nekas avslutas förfrågan på vanligt vis. Om användaren däremot har blivit tilldelad rättighet att använda databasen på det sätt som efterfrågas skickas en fråga till regelsamlingen tillhörande SELinux om huruvida det ska vara tillåtet. Eftersom det inte går att komma åt någon data i databasen utan kontroll i regelsamlingen är säkerhetsmekanismen obligatorisk, det vill säga MAC. Anledningen till att man placerar de nya reglerna tillhörande SE-PostgreSQL i SELinux regelsamling är att reglerna därmed kommer att evalueras på samma sätt och på samma plats som andra objekt i systemet. Det betyder att en användare inte kommer ha större tillgång till data i en databas än han eller hon har i filsystemet, vilket ger resultatet att en av delarna inte har svagare säkerhetskontroll än någon annan och därigenom med större sannolikhet inte blir enklare att komma åt utan tillåtelse.

Tillsammans med databashanterarens säkerhetssamordning med SELinux och införandet av MAC ger SE-PostgreSQL dessutom möjlighet att sätta MAC-rättigheter på enskilda kolumner och tupler. Kolumnerna och tuplerna behandlas som objekt

och blir därmed tilldelade Security Labels som evalueras av operativsystemets kärna. Dessa Security Labels kan av en användare diskret ändras till nya värden i enighet med vad policyn tillåter att användaren ändrar dem till. Mest intressant för diskreta ändringar är finjusteringar av åtkomst med hjälp av MLS/MCS. Det går inte att sätta MLS/MCS-fältet till något man inte själv har åtkomst till.

När det gäller tupler läggs det automatiskt till en extra skrivbar systemkolumn i varje tabell som innehåller tupelns Security Label. Det är därmed lätt att diskret uppdatera den Security Label som används vid evaluering av åtkomst. Det enda som behöver göras är att ändra i den nya kolumnen som om det vore vilken extra kolumn som helst. Den nya kolumnen är dold och listas inte om man inte explicit säger till att man vill visa den (den heter security_label). När en användare vill visa data från en tabell som innehåller tupler han eller hon inte har rättighet att se kommer endast de tupler som är tillåtna att visas (de andra blir stoppade av den obligatoriska säkerhetskontrollen).

Security Labels som hör till andra objekt som exempelvis kolumner, tabeller och databaser sparas i andra externa tabeller. Den extra raden läggs istället till i hjälptabeller som pg_attribute, pg_class och pg_database. Uppdateringar av dessa Security Labels görs därför istället med kommandot ALTER och om man vill se dem får man plocka ut dem ur hjälptabellerna istället [17] [18]. Kommandot för att ändra Security Label på en tabell ser ut på följande vis:

```
=> ALTER TABLE <tabellnamn> SECURITY LABEL TO '<ny security label>';
```

Vill man ändra detsamma på en kolumn lägger man till

```
ALTER COLUMN <kolumnnamn>
```

direkt efter tabellnamnet i kommandot ovan.

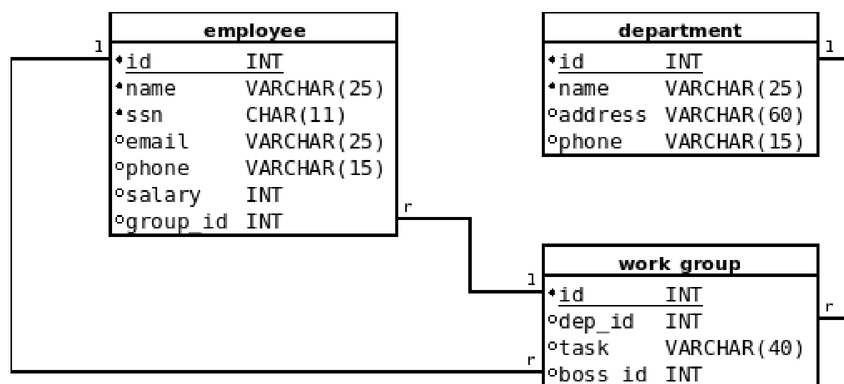
1.3 Syfte

Syftet med den här rapporten är att undersöka hur åtkomsthantering på rad och kolumnnivå påverkar funktionalitet och användbarhet i en databas. Vi har byggt vår databas med SE-PostgreSQL och använt MCS för att hantera åtkomst. Vi har sedan dragit slutsatser om hur hur väl denna åtkomsthantering fungerar och om hur lättanvänd den är. Vi har dessutom undersökt hur man designar en databas med säkerheten i åtanke, t.ex. för att undvika indirekta informationsläckor.

Kapitel 2

Test

2.1 Databasen



Figur 2.1. Databasen som vi använder i våra tester.

För våra tester behöver vi en exempeldatabas där man kan sätta olika behörighet. Vi har valt att skapa en enkel personaldatabas för en fiktiv organisation. Figur 2.1 visar hur databasen är uppbyggd. Databasen är designad för att vi ska kunna genomföra våra tester och är inte medvetet gjord med en speciell organisations behov i åtanke. Databasen består av tre tabeller, en för anställda (employee), en för arbetsgrupper (work_group) och en för avdelningar (department). Tabellen för anställda består av ett id (id), ett namn (name), ett personnummer (ssn), en e-post-adress (email), ett telefonnummer (phone), en lön (salary) samt ett grupp-id (group_id) som länkar ihop den anställda med en arbetsgrupp. Tabellen för Arbetsgrupper består av ett id (id), ett avdelnings-id (dep_id) som länkar ihop arbetsgruppen med en avdelning, en beskrivning av arbetsgruppen (task) samt ett chefs-id (boss_id) som länkar ihop arbetsgruppen med en chef från tabellen anställda. Tabellen för avdelningar består av ett id (id), ett namn (name), en adress (address) och ett

telefonnummer (phone). Alla tabeller har id som primärnyckel.

2.2 Tester

Dessa tester ska demonstrera hur SE-PostgreSQL agerar vid några vanliga användningar av en databas när man har placerat ut Security Labels på tupler och kolumner. I testerna kommer endast MCS att användas för att reglera åtkomsten av objekten, MLS-fältet kommer därför bara att använda sig av en nivå, s0.

2.2.1 INSERT

Scenario från testdatabasen

Alice är en anställd med ansvar för 2 arbetsgrupper och hon har endast tillåtelse att se de grupper hon ansvarar för. Alice blir tilldelad uppgiften att forma ytterligare en arbetsgrupp. Vid kontroll i databasen ser Alice att det finns två arbetsgrupper med identifikationsnummer 1 respektive 2. Alice väljer därför att ge den nya gruppen identifikationsnummer 3.

Problem

Identifikationsnumret är en primärnyckel och måste därför vara unik. När Alice försöker skapa en ny grupp vet hon inte om det redan finns en annan grupp med det valda identifikationsnumret. Frågan är vad som kommer att hända om det redan finns en grupp med identifikationsnummer 3 som Alice inte har tillåtelse att se.

Test

Vi skapar ett konto som representerar Alice med en Security Context som ser ut på följande vis (endast MLS/MCS-delen är viktig, användre, roll och typ är standardvärden för testsystemet):

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c20
```

Om vi endast tittar på kategorierna så betyder det att användaren har tillstånd att se poster som har kategorier inom intervallet 0 till 20. Vi lägger sedan till två tupler med id 1 respektive 2 i work_group-tabellen och ger dem en varsin Security Label med kategorier som Alice har rätt att se, det vill säga som ligger inom intervallet.

```
unconfined_u:object_r:sepgsql_table_t:s0:c10
```

Slutligen lägger vi till ytterligare en tupel i work_group-tabellen och ger den id 3. Denna tupel får en Security Label med en kategori som befinner sig utanför Alice intervall.

```
unconfined_u:object_r:sepgsql_table_t:s0:c30
```

Vi loggar sedan in i databasen med det användarkonto vi skapade för Alice och skickar en SQL-fråga till tabellen för att kontrollera att vi endast ser de första två tuplerna (de med id 1 och 2). Sedan gör vi ett försök att sätta in en ny grupp i databasen med id 3 och ser vad som händer.

Resultat

```
companydb=> SELECT * FROM work_group;
id | dep_id | task  | boss_id
----+--------+-------+---------
 1 |      1 | uppg1 |       1
 2 |      1 | uppg2 |       1
(2 rows)

companydb=> INSERT INTO work_group VALUES(3,1,'uppg3','1');
ERROR: duplicate key value violates unique constraint "work_group_pkey"
DETAIL: Key (id)=(3) aready exists.
```

Från Alice konto kan vi endast se de två rader i work_group som har kategorin 10. När vi försöker lägga till en ny rad med id = 3 stöter vi dock på den dolda tupeln och felmeddelandet indikerar att tupeln redan existerar.

2.2.2 UPDATE

Scenario från testdatabasen

Under en omorganisering på företaget beslutas det att Bob ska bli ansvarig för alla arbetsgrupper på sin avdelning. Bob får uppdraget att uppdatera chefen för alla arbetsgrupper på avdelningen till sitt eget identifikationsnummer. När Bob tar sig en titt i databasen ser han att alla arbetsgrupper tillhör hans avdelning och väljer därför att uppdatera hela tabellen i ett svep.

Problem

Bob har bara tillåtelse att se de grupper som tillhör den avdelning han arbetar vid. Det är fullt möjligt att det finns många fler grupper i tabellen som inte visas när han ställer frågan till databasen. Kommer Bob att lyckas uppdatera tupler han inte känner till och kommer han på något vis märka att det finns fler tupler än de han ser?

Test

Vi börjar med att tillverka ett nytt användarkonto för att representera Bob. Kontots Security Context ser precis likadant ut som Alices sånär som på vilka kategorier han har tillgång till.

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c20.c40
```

Vi loggar på nytt in i databasen, nu med det nya kontot, som fortfarande ser likadan ut som i det förra testet. I tabellen work_group ligger fortfarande 3 tupler varav 1 tillhör kategorin 30 och de 2 andra tillhör kategorin 10. Vid en titt på innehållet borde man därför med Bobs konto endast kunna se en rad (med id = 3).

Vi uppdaterar hela tabellen med boss_id = 2. Vi gör dessutom ett försök att uppdatera en av de rader vi inte har möjlighet att se från Bobs konto. Efter ändringarna loggar vi in med en användare som har tillåtelse att se alla kategorier i databasen och kontrollerar om vi lyckades förändra någon av de osynliga raderna.

Resultat

Inloggad med Bobs användare:

```
companydb=> SELECT * FROM work_group;
id | dep_id | task  | boss_id
----+--------+-------+---------
 3 |      1 | uppg3 |       1
(1 row)

companydb=> UPDATE work_group SET boss_id = 2;
UPDATE 1
companydb=> UPDATE work_group SET boss_id = 2 WHERE id = 1;
UPDATE 0
```

Inloggad som en användare med tillåtelse att se alla kategorier:

```
companydb=> SELECT * FROM work_group;
id | dep_id | task  | boss_id
----+--------+-------+---------
 1 |      1 | uppg1 |       1
 2 |      1 | uppg2 |       1
 3 |      1 | uppg3 |       2
(3 rows)
```

Vi ser att vi från Bobs konto endast lyckades uppdatera de tupler han hade rätt att se. Vidare ser vi att uppdateringarna från hans konto inte visade något tecken på att det egentligen fanns ytterligare två tupler.

2.2.3 SELECT

Scenario från testdatabasen

Man vill att alla anställda på företaget ska ha möjlighet att skicka email till varandra så det känns självklart att alla anställda ska ha tillgång till varandras email-adresser

som finns sparade i tabellen över anställda. Man gör därför tabellen för anställda tillgänglig för alla på företaget. Man vill även att en anställd ska kunna kontrollera vilken avdelning en annan anställd arbetar på.

Problem

Tabellen över anställda innehåller en del privat information såsom exempelvis och lön. Detta attribut vill man att endast löneansvariga och chefen ska kunna se. Vad händer om man döljer detta attribut för samtliga anställda utöver löneansvariga och chef? Är övriga attribut åtkomliga? Syns det att attributet finns även om det inte är åtkomligt? För att koppla en anställd till den avdelning han eller hon arbetar på måste man lägga ihop alla 3 tabeller. Kommer dolda tupler och kolumner fortsätta att vara dolda? Kommer man bli varnad när man lägger ihop tabellerna?

Test

Vi börjar med att göra kolumnen för lön endast tillgänglig för löneansvariga och chefen. Vi antar att löneansvariga utöver chefen har exklusiv tillgång till kategorierna 40-60 och sätter därför kolumnen till kategori 50 i dess Security Label.

```
unconfined_u:object_r:sepgsql_table_t:s0:c50
```

Alice från det första testet har tillgång till kategorierna 0-20 och är ett bra exempel på en anställd som inte borde kunna kolla upp lönen för alla andra anställda på företaget. Via Alices konto försöker vi plocka ut alla anställdas email-adresser. Efter det försöker vi lista hela tabellen och avslutar med att försöka lista endast lön och endast en kolumn som inte existerar för att kontrollera om det blir någon skillnad.

Slutligen vill vi kontrollera huruvida dolda tupler och kolumner fortsätter vara dolda även då vi slår ihop flera tabeller. Detta test utförs genom att vi slår ihop alla 3 tabeller och väljer ut både enskilda tillåtna kolumner och allt i den sammanslagna tabellen (inklusive den dolda salary).

Resultat

```
companydb=> SELECT email FROM employee;
       email
---------------------
charlie@comp.com
bob@comp.com
alice@comp.com
(3 rows)

companydb=> SELECT * FROM employee;
ERROR:  SELinux: security policy violation
```

```
companydb=> SELECT salary FROM employee;
ERROR:  SELinux: security policy violation
companydb=> SELECT car FROM employee;
ERROR:  column "car" does not exist
LINE 1: SELECT car FROM employee;
              ^
```

```
companydb=> SELECT employee.name, department.name
companydb-> FROM employee, department, work_group
companydb-> WHERE employee.group_id=work_group.id
companydb-> AND work_group.dep_id=department.id;
 name    | name
---------+------
charlie | dep1
bob     | dep1
alice   | dep1
(3 rows)
```

```
companydb=> SELECT *
companydb-> FROM employee, department, work_group
companydb-> WHERE employee.group_id=work_group.id
companydb-> AND work_group.dep_id=department.id;
ERROR:  SELinux: security policy violation
```

Det gick att från Alice konto lista alla anställdas email-adresser men vi får ett felmeddelande när vi försöker lista hela tabellen. Samma felmeddelande visas även när vi endast försöker visa kolumnen som vi inte har rätt att se (salary). Då vi försöker visa en kolumn som inte existerar i tabellen (car) får vi ett annat felmeddelande som tydligt visar att den inte finns. I de två sista testen slår vi ihop tabeller och ser att kolumnen salary inte påverkar resultatet så länge vi inte direkt refererar till den.

2.2.4 Aggregerande funktioner

Scenario från testdatabasen

Charlie är ansvarig för sin avdelning och har rätt att se allting som hör till. Han har dock ingen åtkomst till information gällande andra avdelningar. När Charlie listar företagets anställda får han som svar endast de anställda på sin avdelning. Charlie ska sammanställa lite statistik om sin avdelning och vill bland annat kontrollera hur många som arbetar där. För detta ändamål använder han en aggregerande funktion, count, över tabellen employee.

Problem

Kommer en aggregerande funktion att iterera över hela tabellen eller endast över den data som Charlie har rätt att känna till.

Test

Vi antar att alla anställda på avdelning 1 har en Security Label med kategorier inom intervallet 0-80 och att anställda vid andra avdelningar har andra kategorier. Vi börjar med att skapa en användare för Charlie med följande Security Context:

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c80
```

Vi ser sedan till att det finns tre anställda i employee-tabellen (Alice, Bob och Charlie). Tuplerna för Alice och Charlie tilldelas Security Labels med kategori inom intervallet 0-80 och Bob får kategorin 100 som ligger utanför.

Alice och Charlie:

```
unconfined_u:object_r:sepgsql_table_t:s0:c10
```

Bob:

```
unconfined_u:object_r:sepgsql_table_t:s0:c100
```

Vi loggar sedan in som Charlie och kontrollerar att allt fungerar som det ska, det vill säga vi listar innehållet i tabellen och bör bara se Alice och Charlie. Vi räknar sedan antalet personer i tabellen med hjälp av den aggregerande funktionen count samt kontrollerar att det stämmer överens med det antal personer vi såg när vi listade tabellen (2 stycken).

Resultat

```
companydb=> SELECT * FROM employee;
id |  name   |     ssn     |      email       |  phone   | salary | group_id
----+---------+-------------+------------------+----------+--------+----------
 1 | Alice   | 650102-1875 | alice@comp.com   | 08112233 |  24200 |        1
 3 | Charlie | 630304-1965 | charlie@comp.com | 08223344 |  19900 |        2
(2 rows)

companydb=> SELECT count(*) FROM employee;
count
-------
    2
(1 row)
```

När vi listar innehållet i tabellen kan vi som väntat inte se tupeln som beskriver Bob. Funktionen count ger svaret 2 vilket bekräftar att aggregerande funktioner bara körs över de tupler som användaren har rätt att se.

2.2.5 Vyer

Scenario från testdatabasen

För att underlätta användningen av databasen skapas en vy med med ett urval av kolumnerna från tabellen employee. Alice vill lista informationen från vyn.

Problem

Det kan finnas kolumner och tupler med kategorier som inte får visas för användaren i den eller de ursprungliga tabellerna. Kommer vyn att behålla dessa begränsningar på vad användaren ska ha tillåtelse att se?

Test

Vi antar att de tre tuplerna från förra testet är placerade i tabellen över anställda med samma Security Labels. Vidare antar vi att Alice konto har samma Security Context som i det första testet. Det skulle betyda att Alice med kategori-intervallet 0-20 endast skulle kunna se 2 av de 3 tuplerna.

Från en användare med tillgång till alla kategorier skapar vi sedan en vy, short_info, av de 2 kolumnerna name och ssn. Vi ger Alice alla rättigheter till vyn med GRANT och loggar sedan in i databasen med Alices konto.

Från Alices konto försöker vi att lista innehållet i vyn för att se om den dolda tupeln kommer synas ändå eller inte.

Efter det loggar vi återigen in med användaren som har full åtkomst och ser till att kolumnen salary i employee är oåtkomlig för Alice genom att den den har följande Security Context:

```
unconfined_u:object_r:sepgsql_table_t:s0.c50
```

Vi skapar sedan om vyn, nu över de 3 kolumnerna name, ssn och salary. Alice konto ges tillstånd att använda vyn och vi loggar slutligen in på det igen. Den här gången försöker vi både lista allt i vyn och endast de två kolumner som vi vet att Alice har tillstånd att se i originaltabellen.

Resultat

Första inloggningen på Alices konto:

```
companydb=> SELECT * FROM short_info;
 name    |     ssn
---------+-------------
Alice    | 650102-1875
Charlie | 650304-1965
(2 rows)
```

Andra inloggningen på Alices konto:

```
companydb=> SELECT * FROM short_info;
ERROR:  SELinux: security policy violation
companydb=> SELECT name, ssn FROM short_info;
ERROR:  SELinux: security policy violation
```

Från den första inloggningen på Alices konto kan vi utläsa att endast de två tuplerna Alice har tillåtelse att se visas. Den andra inloggningen visar att det inte går att läsa från en vy vilken innehåller kolumner som inte får läsas. Anledningen är att när man anropar en vy utförs först anropet som skapar vyn. Anropet som skapar vyn short_info begär att få visa en kategori som den anropande användaren inte har rätt att läsa och anropet avbryts redan innan det når det yttre anropet, det som användaren faktiskt ser att han eller hon gör.

2.2.6 Användning av "\d"

Scenario från testdatabasen

I PostgreSQL, och därmed även SE-PostgreSQL, finns ett kommando "\d" som används för att visa alla relationer i databasen alternativt alla attribut i en enskild tabell. Kommandot visar dessutom information om främmande nycklar och index.

Problem

Tabeller och kolumner kan vara dolda för en användare. Gäller det även vid användning av "\d"? Avslöjar informationen om främmande nycklar och index någonting om tabellen som det inte är tänkt att användaren ska kunna se?

Test

Vi ser till att vi har en användare med följande Security Context:

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c40
```

För att kunna testa hur väl det fungerar med dolda tabeller och deras kopplingar till andra tabeller väljer vi att dölja work_group. Vi sätter följande Security Label på tabellen:

```
unconfined_u:object_r:sepgsql_table_t:s0:c50
```

Slutligen vill vi testa om index kan avslöja någonting om en kolumn en användare inte har tillåtelse att se och huruvida dolda kolumner syns vid användandet av \d. Eftersom id i employee-tabellen är en primärnyckel finns där redan ett index vi kan använda. Vi tilldelar därför id i employee-tabellen följande Security Label för att testa båda problemen:

```
unconfined_u:object_r:sepgsql_table_t:s0:50
```

När vi har placerat ut våra säkerhetsklassificeringar loggar vi in med den tidigare förberedda användaren och skriver “\d” för att lista alla relationer i databasen. Sedan listar vi kolumnerna i employee tillsammans med information om index och främmande nycklar genom att skriva “\d employee”.

Slutligen gör vi ytterligare ett test genom att byta ut tabellerna mot nya tabeller där alla id-kolumner har datatypen SERIAL. Denna datatyp skapar extra tabeller som kan avslöja viss information om den tabell och det attribut som den är kopplad till.

Vi loggar sedan in med testanvändaren på nytt och testar att lista alla relationer med “\d”.

Resultat

Innan ändringar av datatyper till SERIAL.

```
companydb=> \d
          List of relations
Schema |    Name    | Type  |  Owner
--------+------------+-------+---------
public | department | table | sepgsql
public | employee   | table | sepgsql
(2 rows)

companydb=> \d employee
          Table "public.employee"
 Column  |         Type          | Modifiers
----------+-----------------------+-----------
name     | character varying(25) | not null
ssn      | character varying(11) | not null
email    | character varying(25) |
phone    | character varying(15) |
salary   | integer               |
group_id | integer               |
Indexes:
   "employee_pkey" PRIMARY KEY, btree(id)
Foreign-key constraints:
   "employee_group_id_fkey" FOREIGN KEY (group_id) REFERENCES work_group(id)
Referenced by:
   TABLE "work_group" CONSTRAINT "work_group_boss_id_fkey"
                            FOREIGN KEY (boss_id) REFERENCES employee(id)
```

Efter ändringar av datatyper till SERIAL:

```
companydb=> \d
          List of relations
```

```
Schema |        Name        |   Type   |  Owner
--------+--------------------+----------+---------
public | department         | table    | sepgsql
public | department_id_seq  | sequence | sepgsql
public | employee           | table    | sepgsql
public | employee_id_seq    | sequence | sepgsql
public | work_group_id_seq  | sequence | sepgsql
(5 rows)
```

Vid den första listningen av tabellerna innan byte av datatyper fungerar allt som väntat. Endast de tabeller vi har tillåtelse att se visas. När vi listar informationen om tabellen employee visas endast de kolumner vi har rätt att se. Det visas dock även att det finns ett index på den primära nyckeln id vilken vi inte ska känna till. Dessutom visas kopplingen till en tabell, inklusive attributnamn, som vi inte har rätt att se. Efter att vi har bytt till SERIAL dyker det upp extra sekvenser när vi listar alla relationer.

Kapitel 3

Diskussion

3.1 Säkerhet

När man endast vill dölja informationen som finns i enskilda tupler eller kolumner fungerar SE-PostgreSQL för det mesta väl. Det finns dock några saker som bör tänkas igenom då det är viktigt att vetskap om enskilda kolumners existens hålls hemligt. Det som förmodligen först kommer att märkas av är att SE-PostgreSQL vid försök till åtkomst av dolda kolumner skriver ut ett tydligt felmeddelande som berättar att åtkomsten inte är tillåten. Nackdelen med förfarandet är att man genom att testa att skriva olika kolumnnamn kan lista ut namnen på de dolda kolumnerna. Samtidigt underlättar det vid användande av databasen att faktiskt få ett felmeddelande som förklarar vad som är fel framför att databashanteraren, liknande dess hantering av tupler, låtsas som om kolumnen inte finns (och därmed skickar ett felmeddelande som egentligen inte alls har med felet att göra). Det kan dock därför vara en god tanke att vara noga med att kolumnnamn inte innehåller konfidentiell information. Ett annat problem som kan uppstå med konfidentiella kolumnnamn gäller användandet av SE-PostgreSQLs specialkommando “\d”. När man med detta kommando listar information om en enskild tabell skrivs det ut lite information angående index och främmande nycklar. Informationen består bland annat av namnet på den kolumn man har satt ett index på (tänk på att en kolumn som är primärnyckel även har ett index). Samma sak gäller kolumner som är främmande nycklar med tillägget att de dessutom ger insyn i vilken tabell och kolumn de refererar till. De flesta som använder SE-PostgreSQL kommer dock inte ha något problem med de här punkterna då det vanligtvis är datan i tabellen som är konfidentiell och inte namnen på kolumnerna.

Vissa förhållanden kan dock göra det möjligt att komma åt data i en tabell. Om man använder attribut som tvingar alla tupler att vara unika (som exempelvis en primär nyckel) kan data läcka ut då någon försöker sätta in en tupel med ett unikt attribut som redan finns i tabellen men inte är synligt för användaren. Även främmande nycklar kan ställa till liknande problem när man försöker ta bort en tupel som refereras av en främmande nyckel man inte har tillåtelse att se. Riskerna

med de här säkerhetshålen, som förvisso är ganska små, kan minimeras genom att man använder unika attribut som inte följer något tydligt mönster eller säger något om resten av datan i tabellen[18].

Allt användande av SE-PostgreSQL kräver att det i SELinux regelsamling finns regler som beskriver vad som är tillåtet att göra i databashanteraren. Dessa regler kan ändras för att göra rättigheterna ännu mer specifika. Man kan till exempel bestämma vad olika användartyper har tillåtelse att göra med de tupler eller tabeller de har tillgång till (i våra test var alla användare av samma typ och hade rätt att göra vad som helst med de objekt han eller hon hade tillgång till). Fördelen med reglerna är att man kan göra all åtkomst väldigt restriktiv. Att det är så kraftfullt medför dock nackdelen att det är en komplex uppgift att skriva regler och att det därför är lätt att göra fel.

3.2 Användarvänlighet

När det kommer till användarvänlighet i SE-PostgreSQL, samt databaser i allmänhet, finns det två perspektiv. Dels har vi användare och dels har vi administratörer och programutvecklare.

För en användare bör det inte vara någon större skillnad mellan att använda SE-PostgreSQL och PostgreSQL. Den största anledningen till detta är förstås att användare sällan pratar direkt med databasen. Vanligtvis sitter användaren med ett grafiskt gränssnitt i antingen ett program eller en webbsida. I dessa fall kommer de frågor som ställs till databasen skapas av programmet och användaren slippper bekymra sig om skillnader i systemen. Även i de system där användaren själv får skriva frågor till databasen kommer han förmodligen inte att påverkas i någon större utsträckning. En skillnad man dock kan stöta på är den att man, om man saknar behörighet till en eller flera kolumner, alltid måste ange vilka kolumner man vill plocka ut i en selektion. Att använda “SELECT * ...” blir samma sak som att skriva en selektion på alla kolumner och därmed även de kolumner som man inte har behörighet till. Om man försöker selektera en kolumn som man inte har behörighet till får man ett felmeddelande från SELinux och får inte ut någon data från någon kolumn alls. En användare kan självklart också drabbas av fel som beror på att den som administrerar databasen inte har anpassat sig till SE-PostgreSQL.

För en administratör eller programutvecklare finns det desto mer att tänka på. De saker man bör tänka på vid skapandet av databasen tar vi upp i del 3.3. När man ska administrera en SE-PostgreSQL-databas ska man först och främst tänka på att SE-PostgreSQL bara är ett tillägg. Man får inte glömma att dessutom utnyttja det rättighetssystem som erbjuds av PostgreSQL. Att ge alla användare fulla rättigheter med “GRANT” och sedan förlita sig på att SELinux ska kunna hantera alla situationer är inte en metod som kommer att fungera bra i större system. Förändringar av ett subjekts Security Context, för att tillåta eller neka åtkomst till ett objekt, kan lätt få oväntade effekter såsom att åtkomst till andra objekt än det tänkta förändras. Likaså kan förändringar av ett objekts Security Context leda till

att andra subjekt än det tänkta får förändrad åtkomst. När man använder sig av PostgreSQLs egna rättighetssystem finns ingen risk att man påverkar andra subjekt och objekt än just de man explicit anger vid förändringen.

Om man som administratör vill skapa vyer till användare blir det lite annorlunda mot vanliga PostgreSQL. En vy i PostgreSQL fungerar som en sparad fråga som man kan ge användare rättighet att använda. Eftersom SE-PostgreSQL kräver åtkomstkontroll på varje enskild tabell kan man inte ge en användare rättighet att se en tabell med hjälp av en vy om inte användaren redan har rätt att se den ursprungliga tabellen enligt SELinux-policyn. En sak man också bör tänka på när man skapar vyer är att eftersom en vy fungerar som en sparad fråga måste användaren som ska använda vyn ha rätt att använda alla kolumner som vyn behandlar. En användare kan alltså inte ens selektera de kolumner han har rätt att se om vyn innehåller minst en kolumn som användaren inte har tillåtelse att se. Anledningen till det här beteendet är att en fråga som ställs till vyn egentligen är två frågor. Först ställs den sparade frågan som skapar vyn och sedan ställs användarens fråga på den data som den första frågan returnerade från databasen. När frågan som skapar vyn ställs kontrolleras om användaren har rätt att se allt som vyn efterfrågar och om så inte är fallet avbryts hela anropet och ett felmeddelande skrivs ut. Som användare av vyn kan det vara svårt att förstå varför detta händer och därför bör alltid administratören tänka efter en extra gång när han skapar vyer.

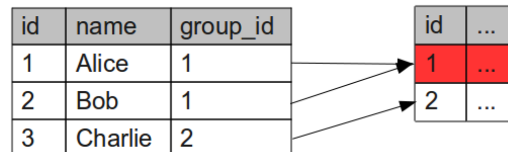
Om man utvecklar program som kommunicerar med SEPostgreSQL finns det ytterligare saker som bör övervägas. Ett Exempel är då man vill lägga ihop ett antal tabeller och skriva ut all information användaren har rätt att se. Betänk att man har två tabeller varav användaren har rätt att se allt i den ena medan några tupler är dolda i den andra. Vidare finns det en koppling mellan varje enskild tupel i den första tabellen och en unik tupel i den andra. När man använder en vanlig join-operation mellan tabellerna kommer alla tupler från den första tabellen som kopplas till en dold tupel i den andra att falla bort, trots att man har rätt att se informationen i den första tabellen. En metod för att slippa bortfallet vore att använda en yttre join. I den resulterande tabellen kommer platserna för de dolda tuplerna fyllas med tomma värden (null) och informationen man har rätt till från den första tabellen kommer vara kvar.

3.3 Design av databas

3.3.1 Normalform och Säkerhet

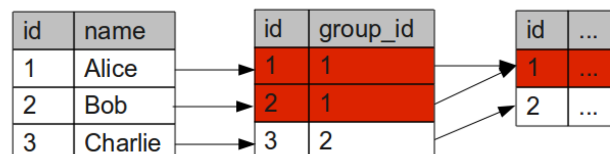
En vanlig strategi när man designar en databas är att försöka normalisera databasen, vanligtvis till tredje normalform. Tredje normalform ger oftast en bra uppdelning på data men den är inte alltid optimal ur en säkerhetssynpunkt. När man normaliserar tänker man nämligen bara på relationen mellan olika data men inte mellan data och användare. En tabell kan därför innehålla både data som användaren har tillåtelse att se och data som användaren inte har tillåtelse att se. Normalisering är ett säkerhetsproblem i de flesta databashanterare men eftersom SE-PostgreSQL kan

hantera rättigheter på rad- och kolumnnivå behöver man inte designa om databasen för det. Ett problem man dock inte blir av med utan att ändra i databasdesignen är indirekta informationsläckor. Med indirekt information menar vi information som inte är direkt synlig för användaren men som man kan listas ut genom att dra slutsatser av den data man har.



Figur 3.1. Databas med en indirekt informationsläcka.

Vi har som exempel de två tabellerna i figur 3.1. Den ena tabellen listar anställda och den andra tabellen listar arbetsgrupper. Alla anställda är synliga men avdelningen med id 1 är dold. Kanske var tanken med att dölja arbetsgruppen bara att dölja den information som finns i tabellen men om man dessutom ville dölja vilka som jobbar där finns det lite mer att önska. Vi får inte ut någon information om den arbetsgrupp som Alice och Bob jobbar på men eftersom vi kan se id:t för alla arbetsgrupper kan vi ändå gruppera anställda efter vilka som jobbar ihop. En strategi som tyvärr inte funkar är att dölja kolumnen group_id. Detta eftersom man då inte kan koppla ihop användare med de arbetsgrupper man har rätt att se. För att lösa detta blir vi tvungna att ändra i vår design. I figur 3.2 har vi lagt till en ny tabell vars enda syfte är att koppla ihop en användare med en arbetsgrupp. Denna design är lik den som uppstår vid en n:n-relation. Om vi vill garantera att den förblir en 1:n-relation blir vi tvungna att sätta kolumnen id i den nya tabellen till att vara unik. I den nya designen har vi möjlighet att blockera information om tillhörighet till vissa arbetsgrupper utan att förstöra kopplingen mellan de användare och arbetsgrupper vi får se. Det finns inget krav på att rättigheterna ska stämma överens mellan den nya tabellen och arbetsgruppstabellen men i de flesta fall är det förmodligen så man vill ha det.



Figur 3.2. Databas där den indirekta informationsläckan är bortbyggd.

3.3.2 Val av primärnycklar

Som datatyp för primärnyckeln i en tabell kan det vara lockande att använda SERIAL. Det är en pseudotyp som kan användas för att automatiskt räkna upp och tilldela en siffra till ett attribut i varje ny tupel. Vad som egentligen händer vid användning av SERIAL är att det skapas en Sequence, en speciell hjälptabell med endast en tupel som används för att spara information om vilket som är nästa tal, som attributet kopplas till. Denna Sequence kommer inte att vara dold oavsett om tabellen eller kolumnen den är kopplad till är dold eller inte. Det går förvisso inte att läsa en Sequence utan att tillstånd lämnas med GRANT men namnet avslöjar information om vad kolumnen och tabellen den tillhör heter. Dessutom följer de automatikt tilldelade siffrorna ett mönster vilket som tidigare nämnt kan vara en nackdel [20]. Som alternativ till att använda datatypen SERIAL skulle man exempelvis kunna använda OID, Object Identifier. Alla systemobjekt tilldelas ett OID vilket betyder att det kan vara svårt att se ett tydligt mönster bland de OIDs de egna objekten tilldelas. Nackdelen är att OID är implementerad som en 32-bitars integer och därför lätt kan ta slut i större databaser. Den säkraste datatypen att använda för primärnycklar är vanliga integers som för användaren inte ser ut att följa något givet mönster. En metod för att uppnå det resultatet vore att skapa primärnycklar genom att hasha innehållet i tupeln.

Kapitel 4

Slutsatser

Man ska aldrig glömma att SE-PostgreSQL i grunden fortfarande är PostgreSQL. Ingen funktionalitet från PostgreSQL har försvunnit eller ändrats på sådant vis att användaren kommer att märka av det alltför mycket. Databashanteraren kan fortfarande till största delen utan problem användas på det gamla vanliga viset utan att någon behöver förändra sin hantering. Några undantag är selektion med “*”, användande av vyer och tillfällen då yttre join kan vara att föredra framför för inre så som diskuterades i avsnitt 3.2.

Det finns ingen mening att använda SE-PostgreSQL utan att använda dess funktioner och även när man använder möjligheten att sätta kategorier på rader och kolumner är det få roller som påverkas nämnvärt. Ett nytt felmeddelande kan dyka upp för alla användare av databasen, nämligen det som skickas när man försöker komma åt kolumner man inte har tillgång till. Felmeddelandet pekar dock inte ut precis vilken data det är som inte är tillåten så det kan upplevas svårt att felsöka, speciellt om användaren har använt “*” vid selektion. Om man inte använder sådana selektionssatser och säkerhetsadministratören inte har gjort några misstag med rättigheterna bör inga svårigheter uppstå.

Att planera vilka kategorier användare ska ha tillgång till och vilken kategori den data de producerar ska få kräver dock i de flesta fall ett omfattande arbete och en god kännedom om organisationen. Det kan exempelvis krävas att en användare ska kunna läsa från en stor mängd kategorier men att vad användaren skriver ska hamna i en specifik kategori som bara några få utvalda användare får läsa. Det är dessutom troligt att ett företag vill ha en egen policy anpassad speciellt för företagets säkerhetsbehov. Att skriva och uppdatera policyn korrekt ger stora fördelar för säkerheten i systemet men eftersom det är ett så kraftfullt verktyg så är det också komplicerat och det krävs mycket noggrann planering för att det ska bli bra.

Litteraturförteckning

[1] *Introduktion till SE-PostgreSQL*, http://wiki.postgresql.org/wiki/SEPostgreSQL_Introduction, 2011-03-24

[2] F. Mayer, K. MacMillan, D. Caplan (2007). *SELinux by Example.* Prentice Hall. s. 5. ISBN 0-13-196369-4

[3] F. Mayer, K. MacMillan, D. Caplan (2007). *SELinux by Example.* Prentice Hall. s. 8. ISBN 0-13-196369-4

[4] B. McCarty (2004). *SELinux.* O'Reilly Media Inc. s. 16. ISBN 0-596-00716-7

[5] National Security Agency. *Bakgrund SELinux*, http://www.nsa.gov/research/selinux/background.shtml, 2011-03-24

[6] F. Mayer, K. MacMillan, D. Caplan (2007). *SELinux by Example.* Prentice Hall. s. 16-32. ISBN 0-13-196369-4

[7] *PostgreSQLs historia*, http://www.postgresql.org/about/history, 2011-02-28

[8] *GRANT i PostgreSQL*, http://www.postgresql.org/docs/8.1/static/sql-grant.html, 2011-02-28

[9] National Computer Security Center (1987). A Guide To Understanding Discretionary Access Control In Trusted Systems. s. 5. Hämtat från: http://www.fas.org/irp/nsa/rainbow/tg003.htm 2011-04-04.

[10] W. Stallings, L. Brown (2008). Computer Security. Pearson Education. s. 711. ISBN 0-13-600424-5.

[11] Department of Defense (1985). Trusted Computer System Evaluation Criteria. s. 59. Hämtat från: http://csrc.nist.gov/publications/history/dod85.pdf 2011-04-04.

[12] Red Hat. Red Hat SELinux Guide: SELinux Users and Roles, http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/4/html/SELinux_Guide/rhlcommon-section-0047.html, 2011-04-04.

[13] W. Stallings, L. Brown (2008). Computer Security. Pearson Education. s. 714. ISBN 0-13-600424-5.

[14] W. Stallings, L. Brown (2008). Computer Security. Pearson Education. s. 304. ISBN 0-13-600424-5.

[15] MLS och MCS, http://wiki.postgresql.org/wiki/SEPostgreSQL_SELinux_Overview#MLS_and_MCS, 2011-04-07.

[16] W. Stallings, L. Brown (2008). Computer Security. Pearson Education. s. 712. ISBN 0-13-600424-5.

[17] SE-PostgreSQLs arkitektur, http://wiki.postgresql.org/wiki/SEPostgreSQL_Architecture, 2011-04-08.

[18] SE-PostgreSQL specifikationer, http://wiki.postgresql.org/wiki/SEPostgreSQL_Specifications, 2011-04-08.

[19] D. Walsh, R. Cooker, T. Bleher. semanage - Linux man page. http://linux.die.net/man/8/semanage, 2011-04-12.

[20] PostgreSQL Documentation - Serial types, http://www.postgresql.org/docs/9.0/interactive/datatype-numeric.html#DATATYPE-SERIAL, 2011-04-13.

Bilaga A

Testsystemet

Operativsystem:	Fedora 14 Desktop Edition Finns att hämta: `https://fedoraproject.org/sv/get-fedora`
Använda paket:	postgresql-8.4.7-1.fc14 (i686) postgresql-libs-8.4.7-1.fc14 (i686) sepostgresql-9.0.1-20101007.fc14 (i686)
SELinux policy:	selinux-policy-targeted-3.9.7-3.fc14 (noarch)

Bilaga B

Redogörelse för samarbete

Under arbetets gång har vi till största delen jobbat tillsammans, d.v.s. suttit tillsammans och diskuterat tester och metoder samt hur vi ska formulera texten i rapporten. Mattias har förvisso skrivit stora delar av bakgrunden medan Andreas ligger bakom mer av diskussionen. All text är dock kontrollerad och uppdaterad av båda två. När det gäller testerna är det Mattias som har jobbat mest med security contexts i systemet medans Andreas har skrivit SQL samt designat databasen.

www.kth.se