

Université du Québec
École de technologie supérieure

Département de génie logiciel et des technologies de l'information

Projet de session :

Sécurité dans les bases de données

Étudiant(s)	Courtois, Thomas Dramé, Yakhoub Servoles, Sébastien
Cours	MTI719
Session	Automne 2011
Professeur(e)	Jean-Marc Robert
Date	07/12/2011

TABLE DES MATIERES

Introduction 4
Problématique 4
Contrôle d'accès basé sur les rôles 6
Introduction 6
Problématique 6
Présentation des principales approches 7
Présentation de MAC et DAC 7
Présentation de RBAC 8
Différentes modélisation de RBAC 10
Première approche : multidimensionnelle 10
Deuxième approche : proposition du NIST 12
Comparaison des deux modèles 14
Comment étendre RBAC ? 16
Introduction 16
Pour faire du MAC 16
Gestion des risques 18
Est-ce que RBAC est indispensable de nos jours ? 21
Les systèmes MAC et DAC 21
Modélisation de RBAC 21
Utilisation dans les bases de données 21
L'avenir de RBAC 22
Conclusion 22
Références 22
Sécurité dans les bases de données orientées objet 23
Introduction 23
Introduction aux Modèles 25
Le premier modèle 25
Le deuxième modèle 34
Concepts clés 39
L'héritage 39
La composition 40
Les versions 42
L'administration des autorisations 43
Agrégation et inférence 44

Conclusion 45
Références 45
Cryptographie dans les bases de données 47
Introduction 47
Cryptographie et base de données 48
Quelle partie crypter ? 48
Gérer les utilisateurs et les clés de cryptage 48
Protéger les requêtes 49
Proposition de solutions théorique 49
Cacher les requêtes à la base de données (PIR = Private Information Retrivial) 49
Ré indexation des données dans la base 50
Le cryptage brut des données 50
Le hachage des données 52
Utilisation d'un catalogue de sécurité 52
Protection contre l'inférence 53
Qu'est-ce qu'une attaque par inférence ? 53
Résultats de cette étude 55
Implémentation d'une solution : CryptDB 56
Le concept 56
Requêtes sur les données chiffrées 57
Gestion des « principaux » 60
Performances 60
Un système prometteur 61
Conclusion 62
Références 63
Conclusion 64

Introduction

De nos jours, les bases de données sont un des éléments sensibles des infrastructures informatiques, puisqu'elles sont déployées massivement dans tout type de structure, des banques aux petites entreprises. En effet, les fichiers utilisés à l'origine pour stocker des données ne sont plus adaptés aux usages actuels. Il est devenu trop compliqué de gérer un flux d'information toujours plus important sans systèmes évolués. Les bases de données apportent une solution à plusieurs problèmes.

Tout d'abord, les données peuvent être centralisées dans un même endroit, et sont accédées depuis des machines clientes à distance. Cela rend disponible l'information en tout temps. Ensuite, les données sont organisées et stockées de manière efficace par les systèmes de gestion de bases de données. Grâce à cette gestion de données, le volume d'information peut devenir très important, sans poser des problèmes d'organisation. L'accès se fait à travers des requêtes écrites dans un langage particulier, comme SQL. L'utilisation d'un langage spécialisé permet de manipuler très facilement les données, puisque des requêtes complexes peuvent être exécutées, aussi bien par un utilisateur que par un programme informatique.

La sécurité de ces systèmes devient alors une priorité, puisqu'il est nécessaire de protéger les données sensibles qui se retrouvent stockées au même endroit. Les personnes malveillantes sont tentées de contourner les règles de sécurité pour voler des informations, les modifier, les supprimer, ou bien encore effectuer des opérations illégales. Le nombre incessant de ces attaques nous force à mettre en place des moyens efficaces et préventifs pour les contrer.

Ces moyens de sécurisation peuvent être de différentes natures. Les moyens techniques mis en place par les développeurs des systèmes de gestion de bases de données permettent d'assurer une première protection efficace. Au-dessus de ces moyens de protection, il est toujours nécessaire et judicieux de mettre en place une politique de sécurité efficace. Celle-ci va dépendre du type de données et d'application à protéger.

Problématique

Dans cette synthèse, nous allons présenter différents aspects de la sécurité des bases de données. Chacun de ces aspects représente un enjeu important pour les entreprises et les organisations qui souhaitent investir dans un système de base de données en garantissant la protection des données, aussi bien au niveau de la confidentialité que de l'intégrité et de la disponibilité.

Les domaines que nous avons choisis ne représentent qu'une partie des points à prendre en compte pour garantir la sécurité. Ils interviennent à des niveaux différents et ne concernent pas les mêmes acteurs. Cependant, ils permettent de mieux appréhender la nature de ces considérations.

Le premier thème choisi est celui du contrôle d'accès. Il s'agit des mécanismes utilisés pour autoriser ou non l'accès à l'information. Ensuite, nous traiterons de la sécurité des bases de données orientées objets. L'objectif est de modéliser des politiques d'accès aux objets qui les peuplent. Enfin, nous allons étudier la cryptographie dans les bases de données, et son implication au niveau de leur infrastructure, de leur organisation et du traitement des données.

Contrôle d’accès basé sur les rôles

Sébastien Servoles

INTRODUCTION

Un des aspects souvent évoqués au sujet de la sécurité des bases de données, et par extension de tout système contenant des informations qui doivent être protégées, est le contrôle d'accès. Il s'agit du mécanisme qui permet de s'assurer que les données et les opérations sur ces données ne sont accessibles que par des utilisateurs autorisés, c'est à dire les utilisateurs qui possèdent des droits suffisants. La confidentialité de ces données est le principal objectif visé par le contrôle d'accès.

Il existe différentes méthodes pour gérer le contrôle d'accès. Elles permettent de définir des politiques variées suivant le type d'application et le contexte dans lequel elles s'instaurent. Elles se basent sur un modèle et des principes différents.

Parmi ces méthodes, trois sont souvent mentionnées lorsqu’on parle de contrôle d'accès : MAC (Mandatory Access Control), DAC (Discretionary Access Control) et RBAC (Role Based Access Control). Ces trois méthodes sont connues et documentées, ce qui permet aux développeurs de solutions informatiques de se baser sur des principes déjà existants pour mettre en place un contrôle d'accès.

Cette synthèse présente plusieurs articles traitant du contrôle d'accès, notamment le contrôle d'accès basé sur les rôles, et proposant une modélisation de cette méthode, ainsi que des pistes pour améliorer la politique de sécurité.

PROBLEMATIQUE

Aujourd'hui, le contrôle d'accès basé sur RBAC est de plus en plus utilisé. Il offre de nombreux avantages par rapport aux méthodes basées sur MAC et DAC. Il permet notamment aux administrateurs de définir de manière plus souple les droits de chaque utilisateur. Les bases de données utilisent ce système depuis plusieurs années.

Bien qu'il propose des concepts intéressants, il n'est pas adapté à toutes les situations, et MAC ou DAC sont parfois préférés. De plus, pour que RBAC puisse s'étende encore à l'avenir, il est nécessaire de définir un modèle pour l'utiliser efficacement. Nous allons donc voir comment RBAC peut être modélisé, comment étendre ses fonctionnalités pour proposer des fonctionnalités présentes dans des systèmes similaires, puis définir si RBAC est prêt à être utilisé massivement.

Dans un premier temps, les principaux modèles seront présentés, avec leurs spécificités et leurs différences. Ensuite, deux approches pour modéliser RBAC seront présentées. Nous

allons également présenter deux initiatives pour améliorer les politiques de sécurité basées sur RBAC. Enfin, nous analyserons les résultats obtenus de ces initiatives afin de mettre en avant le besoin et les avantages d'utiliser RBAC dans un système de gestion de bases de données.

PRESENTATION DES PRINCIPALES APPROCHES

Présentation de MAC et DAC

MAC et DAC sont deux approches pour le contrôle d'accès. Elles se basent sur des concepts différents, et représentent les méthodes traditionnelles pour mettre en place une politique de contrôle d'accès. Elles sont notamment décrites dans le Trusted Computer System Evaluation Criteria, souvent appelé "Orange Book". Il s'agit d'un cahier des charges émit par le gouvernement des États-Unis qui définit des règles d'usage dans la sécurité informatique. Plusieurs niveaux de sécurité peuvent être utilisés.

Mandatory Access Control

MAC (Mandatory Access Control) est une méthode qui se base sur le principe suivant : les propriétaires des objets ne peuvent pas choisir ou modifier les privilèges qu'ils possèdent sur ces objets. Les droits sont donc imposés par le système. Un administrateur a la charge de définir les droits d'accès pour chaque utilisateur. À chaque fois qu'un utilisateur souhaite accéder à une ressource, les privilèges qui lui sont associés sont vérifiés par le système avant d'autoriser l'accès.

Cette méthode est relativement stricte pour un contrôle d'accès, et permet de garantir facilement la confidentialité des données. En revanche, elle peut dans certains cas se révéler inappropriée, puisqu'elle a été initialement été établie dans le cadre de l'accès aux informations du gouvernement des États-Unis. Les utilisateurs peuvent avoir besoin de gérer eux-mêmes ces droits, par exemple dans le cas où ils sont amenés à utiliser fréquemment de nombreuses données.

La sécurité multiniveau (MLS) est un principe qui se base sur MAC. Il consiste à classer des informations selon différents niveaux de confidentialité (publique, classifié, top secret…). Un utilisateur possède un niveau d'habilitation, grâce auquel il peut accéder à ces informations. Cela garantie l'intégrité et la confidentialité des informations classifiées.

Discretionary Access Control

DAC (Discretionary Access Control), par opposition, autorise les utilisateurs à décider des droits sur les objets dont ils sont propriétaire, ou sur lesquels ils disposent d'une autorisation. Ils peuvent également transférer les droits qu'ils possèdent pour d'autres utilisateurs.

Cette gestion est plus souple que MAC. Elle est plus adaptée dans les systèmes où les utilisateurs ont besoin d'avoir plus de contrôle sur les données, ou bien simplement dans les cas où MAC ne peut pas être mis en place.

Un exemple de système reposant sur l'approche DAC est la gestion des droits sous les systèmes Unix. En effet, chaque propriétaire de fichier peut changer le mode d'accès au fichier (en lecture, écriture ou exécution), ou bien transférer ses droits vers un autre utilisateur, en passant par exemple par un groupe. Il possède donc des droits étendus sur ce fichier, afin de décharger l'administrateur de ces tâches.

Modèle Bell-LaPadula

Le modèle Bell-LaPadula est un modèle qui utilise des règles MAC et DAC pour établir une politique de sécurité. Ce modèle se base sur un concept central : « no read-up, no write-down ». Cela signifie qu'un utilisateur peut accéder à des ressources dont le niveau de confidentialité est égal ou plus faible, mais qu'il ne peut écrire vers des ressources de niveau de confidentialité plus faible, mais seulement égal ou supérieur. Cela permet de définir un flux d'information acyclique, puisque les données vont des niveaux de faible confiance vers les niveaux de confiance élevée.

Ce concept met l'accent sur la confidentialité avant tout, et puisqu'il est toujours possible d'écrire vers les niveaux de sécurité les plus avancés, l'intégrité des données n'est pas une priorité.

Présentation de RBAC

Les rôles

Les rôles sont au cœur de la méthode RBAC. Il s'agit d'un ensemble de privilèges ou permissions qui permettent de réaliser des opérations sur des objets. Chaque privilège est un ensemble de droits d'accès vers un objet, qui peut être une donnée ou une opération. Il peut s'agir par exemple, dans le domaine des bases de données, d'un droit en écriture sur une table de données, ou bien encore d'un droit d'exécution pour une procédure.

Les rôles peuvent être comparés aux groupes, dans le sens où, dans les deux cas, un ensemble de permissions est associé à une entité. Cependant, ces deux composants sont différents sur plusieurs points. Tout d'abord, les groupes sont généralement mis en place de manière discrétionnaire, c'est-à-dire que les utilisateurs associés à un groupe peuvent prendre des décisions sur les modes d'accès à un objet. Avec les rôles, les utilisateurs ne peuvent qu'utiliser les permissions qu'ils leur ont été données. Ensuite, la nature même de ces permissions est différente, puisque dans le cas des rôles, elles peuvent être affectées à des fonctions ou des programmes [1].

Un autre concept important amené par les rôles est la gestion d'une hiérarchie. Tout comme certains systèmes utilisés dans l'informatique, tels que les classes dans les langages de programmation orientés objets, les rôles peuvent hériter d'autres rôles afin de faciliter la

modélisation et la mise en place d'une politique de contrôle d'accès. Les rôles « enfants » héritent de toutes les propriétés des rôles « parents », mais peuvent également posséder des propriétés supplémentaires.

Role-Based Access Control

Le contrôle d'accès basé sur les rôles (RBAC) est une alternative aux modèles plus classiques DAC et MAC. En effet, DAC et MAC ne sont pas toujours adaptés aux systèmes actuels. DAC est moins efficace du point de vue de la sécurité, et MAC est principalement destiné au classement de données confidentielles tel que décrit dans l'Orange Book.

RBAC utilise les rôles tels que décrits précédemment pour gérer les autorisations. Pour cela, les utilisateurs d'un système sont associés à un ou plusieurs rôles définis à l'avance. Lorsque ceux-ci souhaitent effectuer une opération, ils vont activer un ou plusieurs rôles pour ouvrir une session. Ils pourront alors effectuer toutes les opérations permises par les rôles de cette session, grâce aux privilèges auxquels ils sont associés. Cette gestion a de nombreux avantages, puisqu'elle ajoute une couche d'abstraction pour l'affectation des utilisateurs aux privilèges sur les ressources.

Le premier avantage est la simplicité au niveau de l'administration. Puisqu'une politique de sécurité est moins susceptible d'évoluer que les utilisateurs associés à un système, il est plus logique de mettre en place dans un premier temps cette politique, puis de gérer les utilisateurs de manière séparée. Avec RBAC, un administrateur peut aisément ajouter des utilisateurs en les associant à différents rôles, puis mettre à jour les associations en ajoutant ou en retirant des rôles.

Ensuite, une politique évoluée pour le contrôle d'accès peut être mise en place plus facilement en utilisant les rôles. L'abstraction qu'ils apportent est plus adaptée à la modélisation de modèles complexes. Des mécanismes supplémentaires permettent d'aller plus loin encore dans la modélisation, avec par exemple la hiérarchie des rôles. Les possibilités offertes par ces mécanismes pour l'établissement d'une politique évoluée sont très grandes.

Enfin, il est possible de modéliser un système basé sur les rôles de manière très avancée. RBAC est un modèle extensible, et un grand nombre de principes peuvent être appliqués avec cette approche. Nous allons voir dans cette synthèse plusieurs propositions de modèles pour l'intégration de fonctionnalités supplémentaires dans un système de ce type.

Au final, un système RBAC peut se révéler efficace dans de nombreux cas de figure, en mettant l'accent sur la confidentialité ou l'intégrité des données selon la politique de sécurité mise en œuvre.

DIFFERENTES MODELISATION DE RBAC

Première approche : multidimensionnelle

Présentation

Le premier article que nous présentons dans le domaine du contrôle d'accès pour cette synthèse est Role-Based Access Control : A Multi-Dimensional View [1]. Les auteurs de cet article proposent un modèle pour mettre en place un système basé sur RBAC, afin de mieux comprendre qu'est-ce que RBAC, quels sont ses possibilités et ses limites.

Tout d'abord, il est rappelé dans cet article quel est le principe de RBAC, et pourquoi il représente une alternative intéressante face à MAC et DAC. L'accent est mis sur les différences entre un modèle RBAC et une implémentation. Les groupes, utilisés par exemple avec les systèmes Unix, peuvent dans certains cas être assimilés à des rôles. Il est vrai que le comportement final peut être similaire, mais les auteurs insistent sur la fait que ce n'est pas forcément une bonne approche, dès lors que l'objectif de base de ces deux concepts n'est pas le même. En effet, RBAC couvre en général beaucoup plus de notions, ce que l'on peut voir dans les nouvelles implémentations de RBAC dans les logiciels récents.

Afin de définir quels sont les principaux champs couvets par RBAC, les fonctionnalités ont été séparées en plusieurs dimensions. Chacune de ses dimensions se distingue des autres car elle peut être développée de manière plus ou moins avancée selon la politique de contrôle d'accès désirée.

Les dimensions considérées par les auteurs de l'article sont les suivantes :

a) La nature des privilèges et les permissions

Cette dimension consiste à définir la nature des privilèges, c'est-à-dire les types d'opération que les utilisateurs peuvent effectuer sur les données. Elle peut être plus ou moins avancée selon le degré de granularité souhaité. Il peut s'agir par exemple d'un droit de lecture sur un type de données, ou bien encore sur certaines données en particulier.

b) Hiérarchie des rôles

Cette dimension est un autre aspect important de RBAC. Il s'agit de la manière dont les rôles peuvent hériter les uns des autres, comme nous l'avons vu précédemment. Plusieurs structures peuvent être utilisées pour implémenter la hiérarchie, de la structure en arbre à une structure qui gère l'héritage multiple.

c) Association des utilisateurs

Cette dimension traite de la manière utilisée pour associer les utilisateurs aux rôles. Plusieurs questions doivent être résolues avant de traiter ce point. Tout d'abord, il faut

définir qui a le droit et la responsabilité de fixer ces associations. Cela va dépendre de la souplesse du système et du niveau de sécurité voulu. Ensuite, il est nécessaire de prendre en compte des limitations lors de l'affectation d'un utilisateur à un rôle. Pour des raisons de sécurité, il est nécessaire de séparer certains rôles ayant des fonctions complémentaires ou contraires. Cette séparation est expliquée plus en détails avec le prochain article.

d) Affection des privilèges et des permissions

L'implémentation de cette fonctionnalité permet d'associer les rôles à des permissions. Tout comme l'association des utilisateurs, différentes méthodes peuvent être utilisées, notamment au niveau de la responsabilité de cette affectation. Des contraintes similaires au moment de l'affectation peuvent être mises en place.

e) Usage des rôles

Cette dimension traite de l'utilisation des rôles. Elle définit notamment si un utilisateur peut activer plusieurs rôles en même temps. Cela va dépendre de la politique à mettre en place, et de la nature même des rôles définis.

f) Evolution des rôles

L'évolution des rôles est un aspect important à prendre en compte. Il s'agit de définir comment les rôles peuvent et vont évoluer au cours du temps, comment les mettre à jour, les associer…

g) Attributs des objets

Cette dernière dimension est la définition d'attributs pour les objets. De la même manière que les utilisateurs sont regroupés grâce aux rôles, les objets peuvent être classés selon certains critères, comme leur utilité ou le contexte dans lequel ils peuvent être utilisés.

Analyse

Le travail effectué par les auteurs de cet article vise à établir des règles simples et compréhensibles pour la mise en place d'un modèle RBAC, au moment où cette approche commence à prendre de plus en plus d'importance, dans les années 90. Cette initiative est donc utile dans ce contexte, puisque peu de documentation était alors disponible, et encore moins de modèle stable définissant avec précision les différents points à prendre en compte.

Les différentes dimensions sont présentées de manière claire. Elles couvrent un grand nombre de fonctionnalités que l'on retrouve dans beaucoup de systèmes RBAC, ainsi que des fonctionnalités avancées telles que la hiérarchie des rôles.

Il n'est pas évident de s'assurer que tous les domaines sont couverts par cet article. Les auteurs en ont conscience, et admettent qu'il ne s'agit que d'une proposition, et non d'un modèle définitif. Il est possible d'ajouter d'autres dimensions à celles qu'ils ont repérées, ce que les lecteurs sont d'ailleurs invités à faire.

S'agissant d'un travail intervenant seulement quelques années après les premiers usages de RBAC, les dimensions proposées ne sont pas regroupées ou classées selon des critères communs. Elles sont toutes définies les unes après les autres, et se différencient par un objectif différent. Les développeurs souhaitant utiliser ce modèle pour implémenter le contrôle d'accès peuvent donc trouver un manque de cohésion. De plus, les compétences requises pour chaque partie ne sont pas définies. Un développeur souhaitant implémenter ce modèle ne connait pas forcément avec précision les politiques à mettre en place par exemple. Il n'y a pas non plus d'ordre à respecter pour l'implémentation.

Ces inconvénients empêchent donc ce modèle de servir de référence à long terme pour la modélisation de RBAC. Cependant, l'initiative est louable puisqu'elle intervient à un moment où le manque de définition et de modèle précis de RBAC commence à être problématique.

À partir de cet article, l'implémentation de RBAC est facilitée, et la modélisation d'un modèle plus complet peut utiliser ces dimensions comme première base.

Deuxième approche : proposition du NIST

Introduction

Le deuxième article est une proposition du NIST pour l'établissement d'un standard RBAC : Proposed NIST Standard for Role-Based Access Control [2]. Le NIST est une agence américaine pour le développement de nouvelles technologies et de nouveaux standards en accord avec le monde de l'industrie. Les motivations d'établissement d'un standard sont proches de celles des auteurs du premier article, à savoir l'établissement d'un ensemble de règles et de bonnes pratiques pour implémenter un contrôle d'accès.

Cet article intervient plusieurs années après le premier. Entre temps, RBAC a été de plus en plus utilisé dans les systèmes, et il devient de plus en plus important d'établir une norme pour les futurs développements. L'approche est donc différente du premier article, et se veut ici structurée, en essayent de couvrir tous les aspects de RBAC. Les méthodes utilisées dans l'industrie permettent de faciliter la mise en place d'un modèle, puisque certaines règles sont déjà admises et implémentées.

Dans un premier temps, les auteurs rappellent le principe du contrôle d'accès basé sur les rôles. Ensuite, la mise en place du standard est découpée en deux parties : le modèle et les spécifications.

Le modèle

Le modèle est un ensemble de règles qui décrivent plusieurs ensembles d'éléments d'un système basé sur RBAC. Parmi ces ensembles, on trouve :

a) Le cœur de RBAC

Cet ensemble de fonctions regroupe les principales fonctionnalités de RBAC, telles que la définition et la gestion des rôles, des utilisateurs et des permissions, ainsi que l'affectation de ces entités entre elles. Il s'agit du seul composant obligatoire lorsque l'on souhaite implémenter RBAC.

b) La gestion de la hiérarchie des rôles

Le second ensemble est la gestion de la hiérarchie. Comme nous l'avons vu précédemment, la hiérarchie est un aspect important de RBAC puisqu'il permet de définir des rôles de manière plus précise et complète, en définissant des rôles « parents » et des rôles « enfants » qui héritent des propriétés et des privilèges.

Deux types de hiérarchie de rôles sont définis par les auteurs : la hiérarchie RBAC générale, qui inclus la plupart des concepts de l'héritage tels que l'héritage multiple, et la hiérarchie RBAC limitée, qui impose des restrictions telles que la structure de l'arbre hiérarchique.

c) Séparation statique des rôles par devoir

Le troisième ensemble est la séparation statique des rôles en fonction de leurs responsabilités. Si deux rôles proposent des droits contradictoires, c'est-à-dire qu'il est incohérent pour un utilisateur de posséder les deux rôles en même temps, alors il ne devrait pas être possible de l'affecter à ces deux rôles. Pour établir cette règle, il est nécessaire de définir un ensemble de rôles parmi lesquels un utilisateur ne peut être associé qu'à un certain nombre.

Deux types de séparation statique existent : la séparation statique classique et la séparation statique avec la hiérarchie des rôles. Dans le cas de la séparation des rôles avec la hiérarchie, le comportement est similaire à la différence près que les rôles hérités sont également pris en compte, de la même manière que les rôles dont ils sont hérités.

d) Séparation dynamique des rôles par devoir

Tout comme la séparation statique des rôles, la séparation dynamique vise à éviter à ce que les utilisateurs soient affectés à plusieurs rôles contradictoires. La différence entre les deux approches se situe dans la manière utilisée pour placer des contraintes d'affectation. Dans le cas de la séparation dynamique, cette décision est prise au moment où l'utilisateur active un rôle auquel il est affecté. La liste des rôles actuellement utilisés par l'utilisateur est vérifiée afin d'accepter l'opération d'ouverture de session.

Les spécifications

Les spécifications définissent les objectifs fonctionnels requis pour chaque ensemble. Elles sont regroupées en trois catégories.

a) Les opérations administratives

Ces opérations permettent de créer et de maintenir les éléments de chaque ensemble. Dans le cas des fonctionnalités du cœur de RBAC, il s'agit de créer des utilisateurs et des rôles, puis de les associer entre eux. Pour la gestion de la hiérarchie, il faut savoir si un utilisateur peut être associé ou retiré d'un rôle en tenant compte des relations plus complexes entre les rôles. Pour la gestion de la séparation des rôles, il faut pouvoir contrôler et s'assurer que les utilisateurs soient affectés à des rôles dans les limites permises.

b) Fonctions de revues

Les fonctions de revues permettent de visualiser les différentes entités du système du point de vue de l'administrateur et de l'utilisateur concerné. Les fonctions définies dans cette catégorie sont donc principalement destinées à retourner des informations.

c) Les fonctionnalités au niveau du système

Cette troisième partie concerne les fonctions techniques utilisées par le système pour permettre aux objets de pouvoir être créés et manipulées. Cela concerne par exemple la création d'un environnement d'exécution, ou session, pour l'activation d'un rôle par un utilisateur, ou bien encore l'ajout d'un rôle dans une session active.

Une liste de spécifications plus détaillées est fournie par les auteurs de l'article en annexe. Cette liste a été mise en place afin de fournir un maximum d'éléments pour l'implémentation d'un contrôle d'accès au sein d'un système.

Analyse

L'un des objectifs du NIST est de proposer des standards en adéquation avec les pratiques utilisées dans le monde l'industrie. RBAC est utilisé depuis plusieurs années dans de nombreux systèmes, ce qui permet de synthétiser et de structurer toutes les pratiques connues. L'approche du NIST est donc intéressante, puisqu'elle permet de se baser sur un ensemble de règles structurées pour établir un système basé sur RBAC. Cette vision permet en outre de mieux définir le rôle de RBAC.

Comparaison des deux modèles

Les deux articles que nous avons présentés proposent un moyen de modéliser un système RBAC. Ils interviennent chacun dans un contexte particulier. Dans le cas du premier article, il s'agit d'établir les bases de la modélisation de RBAC alors que son utilisation est de plus en plus courante. Les aspects traités sont donc moins bien organisés que dans le cas du

deuxième article, paru plusieurs années après. Cette première différence étant naturelle, nous n'allons pas détailler les différences au niveau de la structure et de l'organisation, mais au niveau de la pertinence des points abordés.

Dans le cas du premier article, les dimensions mises en avant couvrent à la fois les fonctionnalités de base de RBAC et les fonctionnalités plus avancées et peu implémentées. On ne retrouve pas tous les points abordés par le second article, comme la séparation des rôles.

Avec le deuxième article, les mécanismes sont décrits avec plus de précision. Les modules présentés sont moins nombreux puisqu'ils regroupent beaucoup plus de fonctionnalités. Au final, celles-ci couvrent plus de domaines. De plus, les spécifications donnent une indication plus précise des fonctionnalités finales offertes aux utilisateurs.

La proposition du NIST est donc, à l'heure actuelle, l'un des modèles les plus efficaces et les plus complets pour implémenter un système basé sur RBAC. Cependant, nous allons voir que cette modélisation ne répond pas à tous les besoins, puisque des propositions supplémentaires ont été publiées afin d'étendre les fonctionnalités présentées ici.

COMMENT ETENDRE RBAC ?

Introduction

RBAC permet de réaliser des architectures souples et extensibles, et de définir de nouveaux usages. Les deux prochains articles proposent d'étendre les capacités de systèmes RBAC afin d'intégrer des fonctionnalités présentes dans d'autres systèmes.

Pour faire du MAC

Introduction

Comme nous l'avons vu précédemment, MAC (Mandatory Access Control) permet d'établir une politique de sécurité robuste et efficace dans certaines situations qui nécessitent un haut degré de confidentialité.

L'article « Modeling Mandatory Access Control in Role-Based Security Systems » [3] est une proposition de modèle pour mettre en place une gestion de la sécurité conforme aux concepts de MAC avec un système basé sur RBAC, afin d'améliorer le degré de confidentialité des données sensibles. Pour cela, plusieurs aspects doivent être pris en compte.

Modélisation de MAC dans un système RBAC

Tout d'abord, les auteurs rappellent les concepts clés de MAC et RBAC. Les deux systèmes ont des objectifs différents, mais la souplesse de RBAC permet de réaliser de nombreuses politiques différentes grâce à une couche d'abstraction supplémentaire pour l'association des privilèges avec les utilisateurs.

Plusieurs termes sont ensuite expliqués afin de définir les bases nécessaires à l'établissement d'un modèle MAC. Le sous-contexte est ainsi introduit comme l'ensemble des ressources accessibles par un unique privilège. Cette notion est importante puisque c'est sur elle que vont reposer les futurs concepts. Ainsi, le contexte est, par extension, l'ensemble des sous-contextes auxquels un utilisateur a accès via un rôle. Concrètement, il s'agit de l'ensemble de l'information et de ses modes d'accès accessibles à un utilisateur à partir d'un rôle, plusieurs contextes pouvant alors être attribués à l'utilisateur suivant les rôles qu'il utilise au cours d'une session.

Afin de modéliser MAC dans un système RBAC, le modèle MLS (Multi Level Secure) a été choisis. Il s'agit de définir un degré de confidentialité pour chaque ressource accessible. Dans un système RBAC, ce sont les contextes qui sont évalués en fonction de leur confidentialité. L'objectif est alors de s'assurer que les contextes ne sont pas contradictoires à la politique MAC, c'est-à-dire que leur champ d'action est restreint selon des règles strictes d'accès, aussi bien en lecture qu'en écriture.

L'une des notions les plus importantes apportée par MAC est le flux d'information acyclique. Un flux d'information est un chemin logique emprunté par des données lorsqu'un utilisateur exécute un programme ou une opération informatique. Dans un environnement ouvert et non-sécurisé, un tel flux relierai toutes les sources d'information avec les autres, puisque chacune d'entre elle pourrait supporter des opérations de lecture et d'écriture à partir de n'importe quelle autre source.

Dans un environnement MAC, le flux d'information est limité grâce aux deux règles définies dans le modèle de Bell-LaPadula : « no read-up and no write-down ». Les informations contenues dans un objet ne peuvent se déplacer que dans un sens, celui des objets dont le degré de confiance est plus élevé.

Les auteurs présentent ce mécanisme comme un moyen de protection efficace contre les chevaux de Troie. Il s'agit de programmes malveillants qui s'exécutent avec les permissions de l'utilisateur courant, dans le but d'exécuter des opérations ou de voler des informations. Dans un contexte MAC, les informations ne pourront pas être dévoilées, puisqu'un utilisateur ne pourra pas accéder en écriture à un niveau de sécurité faible en même temps qu'à un niveau de sécurité élevé en lecture.

Afin de mettre en place une politique MAC dans un système RBAC, les auteurs préconisent un ensemble de règles pour ce système, en se basant sur les notions abordées auparavant. Ainsi, il est nécessaire de s'assurer que les utilisateurs accèdent aux objets en accord avec la définition des rôles, mais surtout que le flux d'information reste à tout moment acyclique, selon la définition donnée lors de la mise en place de la politique de sécurité.

Au final, un ensemble de contraintes a été compilé afin de garantir le respect du flux d'information. Ces contraintes couvrent tous les cas d'utilisation pour la modélisation d'un système simple. Tout d'abord, il s'agit de vérifier que le flux est bien conforme à la politique de sécurité mise en place. Ensuite, les utilisateurs ne doivent accéder à une ressource que dans un mode d'accès autorisé et explicitement associé via un rôle et un privilège. Si le contexte est conforme à la politique de flux, alors l'utilisateur est libre d'effectuer toutes les opérations auxquelles il a accès. Enfin, les règles d'écriture et de lecture ne doivent jamais rompre le principe du flux acyclique.

Analyse

Dans cet article, les auteurs ont voulu démontrer qu'il est possible d'améliorer la sécurité des systèmes RBAC en imitant les caractéristiques de MAC. Pour cela, après avoir rappelé les caractéristiques de chaque approche, un lien a été établi, à travers la notion du contexte, entre les niveaux de sécurité présents dans les systèmes de sécurité multiniveau (MLS) et l'affectation des rôles aux utilisateurs. A partir de cette approche, il a été possible de définir les règles et les contraintes nécessaires pour appliquer une politique MAC.

Ce modèle étend les caractéristiques établies par les deux premiers articles, en modélisant l'affection des rôles d'une manière différente, basée sur les flux d'information. Cela montre

que la modélisation de RBAC n'est pas évidente et établie de manière définitive, puisqu'elle est susceptible de toujours évoluer selon les objectifs de chaque politique.

Cette initiative aura donc permis de montrer qu'il est possible d'étendre les possibilités offertes par un système RBAC, ce qui conforte la position de cette approche pour une utilisation à long terme dans les systèmes informatiques actuels et futurs.

Gestion des risques

Introduction

Le dernier article que nous présentons dans le domaine du contrôle d'accès est « Using Trust and Risk in Role-Based Access Control Policies » [4]. Il s'agit d'une nouvelle proposition pour étendre les concepts de RBAC vers un composant utilisé dans certains systèmes : la gestion de risque. Cette gestion représente un nouveau principe pour le contrôle d'accès : TBAC (Trust-Based Access Control).

Nous allons tout d'abord présenter le principe de la gestion de risque, avant de synthétiser les méthodes utilisées par les auteurs de l'article pour implémenter ce mécanisme dans un système RBAC.

Principe de la gestion de risque

Lorsqu'une opération doit être effectuée, un gestionnaire de contrôle d'accès doit analyser la requête et décider s'il s'agit d'une opération légale ou non. Pour cela, plusieurs solutions : la requête peut être autorisée explicitement par un administrateur dans le cadre d'une politique d'accès, auquel cas l'opération peut se dérouler sans problèmes, ou alors la décision doit être calculée suivant diverses variables. Dans ce cas, nous parlons de gestion de risque, puisque le principal rôle de la fonction qui va analyser ces variables sera de prendre une décision en tenant compte du risque que représente l'opération.

Pour calculer le risque, il faut tout d'abord récupérer certaines informations, puis les compiler afin d'obtenir un résultat. Le risque est notamment calculé à partir du degré de confiance et du coût d'une opération.

Le degré de confiance est calculé à partir de plusieurs variables. Tout d'abord, l'utilisateur qui souhaite effectuer l'opération va devoir prouver qu'on peut lui faire confiance, en donnant des preuves de confiance. Ces preuves peuvent être de différentes natures, comme des recommandations de la part d'une source de confiance. La provenance de ces preuves a une influence sur leur légitimité. D'autres informations sont également utilisées pour déterminer le degré de confiance, comme la légitimité des actions déjà effectuées par l'utilisateur, le contexte (la confiance accordée ne sera pas la même selon la nature et le domaine de l'opération visée), ou bien encore le moyen utilisé pour l'authentification (mot de passe, clé publique…).

Le coût d'une opération est calculé en fonction du coût de chaque fonction qu'il comprend. Chacune d'entre elle est analysée séparément. Au final, le degré de confiance est comparé au coût de l'opération. Si le degré de confiance est supérieur, alors l'opération est autorisée par le gestionnaire de contrôle d'accès.

Implémentation de la gestion de risque

Cet article fait partie d'un projet plus vaste visant à mettre en place un framework pour la mise en place de politiques efficace basées sur RBAC : OASIS (Open Architecture for Secure Interworking Services). Ce framework s'appuie sur un langage particulier permettant de paramétrer tous les éléments. Avec le projet OASIS, les auteurs se sont intéressés à de nombreux aspects du contrôle d'accès, dont la gestion de risque.

Après avoir introduit le principe de la gestion de risque, un deuxième framework est présenté dans l'article : SECURE (Secure Environments for Collaboration among Ubiquitous Roaming Entities). Ce framework a pour objectif de proposer un moyen pour mettre en place un contrôle d'accès basé sur la confiance. Il intègre les différents mécanismes présentés précédemment.

L'une des forces du projet OASIS est son ouverture. Son langage permet de définir des politiques très avancées. Ce langage a pourtant des limites, et ne permet pas de définir tous les points présents dans SECURE. Il est donc nécessaire de faire appel à une ressource externe pour étendre le modèle OASIS.

Un exemple : The Grid

Afin d'illustrer les concepts décrits au cours de l'article, les auteurs prennent pour exemple la validation des opérations dans le cas d'un système de machines « en grille », c'est-à-dire qu'un ensemble d'ordinateurs constitue une certaine quantité de ressource qui peux varier suivant les besoins. Il s'agit d'un système équivalent au « Cloud Computing » qui, aujourd'hui, s'étend dans de nombreux domaines.

Plusieurs cas d'utilisation sont décrits dans le cadre de l'utilisation de services de « la grille » : le partage d'informations de recherche avec plusieurs collaborateurs, le calcul distribué d'un grand nombre de données, et la mise à disposition d'un article controversé. Chacun de ces cas représente une utilisation spécifique des services en termes de ressource à allouer, de disposition géographique, et de confidentialité, intégrité et disponibilité de l'information.

La gestion des risques intervient ici au moment de la définition des droits de chacun vis-à-vis des données. Chaque action, que ce soit pour la publication ou la lecture des informations, est associée à un certain niveau de risque selon différentes variables. Les utilisateurs qui souhaitent exécuter l'une de ces actions sont associés à des taux de confiance. Ces valeurs sont analysées selon la politique établie, afin de garantir ou non l'accès.

Analyse

Dans cet article, les auteurs explorent une nouvelle piste pour étendre le concept du contrôle d'accès vers de nouveaux domaines. Celle-ci rentre dans le cadre d'un projet plus important qui vise à établir des politiques avancées pour le contrôle d'accès : OASIS. Ce projet a pour objectif de proposer un moyen d'établir des communications sécurisées entre des services distribués. La flexibilité est un des points les plus importants de ce projet. Ainsi, il est possible d'étendre les fonctionnalités de base à travers un langage dédié ou des extensions.

La gestion de risque devient un élément important à prendre en compte aujourd'hui. En effet, les services proposés en entreprise et sur internet sont de plus en plus nombreux, et concernent toujours plus de personnes et de machines, avec notamment l'évolution des nouveaux terminaux mobiles tels que les téléphones intelligents et les tablettes. Cette rapide évolution demande des politiques d'accès plus flexibles, rapides et efficaces. La solution dans ce contexte est de laisser les applications prendre plus de responsabilité pour ce type de tâche.

Avec cet article, il est démontré que ces objectifs peuvent être atteints lorsque les bases fondatrices sont définies avec précision. Ici, une base solide et extensible, associée à un outil puissant pour la gestion de risque, ont permis de mettre en place une politique efficace dans plusieurs cas de figure d'actualité.

EST-CE QUE RBAC EST INDISPENSABLE DE NOS JOURS ?

Les systèmes MAC et DAC

Les systèmes MAC et DAC sont toujours utilisés dans de nombreux systèmes de nos jours. Bien que RBAC permette, dans certains cas, de reproduire le comportement de l'un de ces deux systèmes, cela reste plus complexe à mettre en place. Les petits systèmes, non critiques, n'ont donc pas toujours besoin d'utiliser cette approche.

On peut donc dire que les traditionnelles MAC et DAC sont toujours utiles de nos jours, même s'ils ne suffisent pas toujours. De plus, il est indispensable de bien comprendre leurs principes pour modéliser un système plus complet, basé par exemple sur RBAC.

Modélisation de RBAC

Le travail effectué par le NIST sur RBAC est une première référence pour la modélisation. Globalement, les techniques utilisées sont bien connues et documentées. RBAC est suffisamment mature pour une utilisation globale. L'utilisation massive de ce système depuis de nombreuses années montre également qu'il est robuste et prêt à être utilisé dans toutes les branches de l'industrie.

La question peut se poser de savoir si on peut réellement définir un modèle exhaustif pour RBAC alors même que diverses tentatives pour étendre ses fonctionnalités naissent régulièrement. Si l'objectif est de définir un socle pour modéliser les concepts de base de RBAC, alors les initiatives telles que celle du NIST peuvent être admises. Par contre, il est difficile, pour le moment, de définir toutes les fonctionnalités permises par cette architecture.

Utilisation dans les bases de données

En ce qui concerne les bases de données, la plupart d'entre elles utilisent aujourd'hui un système basé sur RBAC pour contrôler l'accès aux données. La politique de sécurisation des données est un aspect très important pour les systèmes de gestion de bases de donnes, et c'est pourquoi l'abstraction permise par l'utilisation des rôles est très bien intégrée dans ces systèmes.

Les contrôles d'accès dans les bases de données intègrent également des fonctions avancées pour se protéger des attaques, comme une protection contre les attaques par brute-force, les contraintes sur les règles de définition des mots de passe, les certificats de sécurité ou bien encore l'authentification sécurisée. Ces mesures de protection permettent aux bases de données d'assurer une sécurisation efficace en ce qui concerne le contrôle d'accès. Cependant, d'autres aspects doivent être pris en compte si l'on veut appliquer une politique de sécurité générale pour une base de données.

L'avenir de RBAC

Plusieurs travaux ont prouvés que le contrôle d'accès basé sur les rôles est suffisamment souple pour introduire de nouvelles politiques. Cette extensibilité est un indicateur sur la pérennité de cette méthode. Il reste probablement des nouvelles fonctions à mettre en place à partir de ce qui existe aujourd'hui. Cette approche est donc positionnée pour durer un certain temps.

CONCLUSION

Dans cette synthèse, nous avons étudié différents articles traitant du contrôle d'accès, et plus précisément du contrôle d'accès basé sur l'utilisation des rôles.

Deux de ces articles établissent un ensemble de règles pour mettre en place ce type de contrôle, alors que deux autres articles introduisent des fonctionnalités supplémentaires avec RBAC pour répondre à des besoins spécifiques.

Au final, nous avons vu que RBAC est aujourd'hui une approche suffisamment développée et sécurisée pour proposer une alternative intéressante aux traditionnels MAC et DAC, et ce dans un grand nombre de systèmes gérant des données confidentielles.

Dans le cas des bases de données, le contrôle d'accès est un des éléments les plus importants à prendre en compte pour la politique de sécurité, mais ce n'est pas suffisant. Nous allons voir dans les prochaines parties quelles sont les autres problèmes à traiter lorsqu'on souhaite se protéger de toutes les menaces qui les entourent.

RÉFÉRENCES

[1] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, Charles E. Youman : Role-Based Access Control: A Multi-Dimensional View

[2] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn et Ramaswamy Chandramouli : Proposed NIST Standard for Role-Based Access Control

[3] Matunda Nyanchama et Sylvia Osborn : Modeling Mandatory Access Control in Role-Based Security Systems

[4] Nathan Dimmock, Andras Belokosztolszki, David Eyers, Jean Bacon et Ken Moody : Using Trust and Risk in Role-Based Access Control Policies

Sécurité dans les bases de données orientées objet

Yakhoub Dramé

INTRODUCTION

Une base de données orientée objet est un ensemble d'information stocké dans un dispositif informatique sous la forme d'objets persistants. Autrement dit, les bases de données orientées objet permettent de prolonger leur durée de vie. Les objets sont des concepts ou entités possédant une structure interne, des attributs et un comportement représenté par des méthodes. Chaque objet est dérivé d'une classe servant de modèle et il hérite de ses caractéristiques. Les objets sont créés lors de l'exécution des programmes, puis ils disparaissent à la fin de cette exécution. On retrouve notamment cette notion d'objet dans les langages de programmation orientés objet tel que le langage JAVA.

Une base de données orientée objet allie donc la puissance de la programmation objet à celle de la gestion de base de donnée. Elles sont donc de très bons outils qui supportent les applications et à un autre niveau leur design et leur développement tout en garantissant une sécurité et protection de données. En ce qui regarde leur sécurité, beaucoup de modèles ont été proposés récemment dans la recherche en matière de sécurité dans les bases de données orientées objet. En principal le réel combat que connaît cette gestion est la gestion de l'accès à ces données, autant à des utilisateurs autorisés à y accéder qu'a des utilisateurs non autorisés. Contenant des informations plus ou moins sensibles, les bases de données doivent être aptes à partager d'une façon réfléchie les données parmi les utilisateurs.

Certains modèles se concentrent sur la protection des informations pour des objets à un niveau de sécurité unique ou des *single-level objects*. Cependant certaines situations amènent les objets à posséder des variables d'instance à plusieurs niveaux de sécurité. Pour atteindre ce résultat ces modèles doivent supporter des objets à plusieurs niveaux de sécurité. Dans cette synthèse, nous nous penchons sur ces deux approches.

En ce qui touche la première approche, les objets doivent être les unités centrales des autorisations. Autant dire que les objets sont les cibles et sources d'informations dont on doit contrôler l'accès. En outre, le modèle doit avoir différents niveaux de granularité : les autorisations doivent s'accorder aux objets, à leur classe ou même à l'ensemble de la base de données. Un modèle d'autorisation avait déjà été défini par le framework Orion system, mais celui-ci ne contient pas une spécificité important : la définition d'autorisations qui découlent du contenu même de l'objet.

Le modèle proposé sera un modèle révisé de celui du framework d'Orion system avec quelques simplifications, à savoir, la proposition de deux modalités pour l'administration d'autorisation et une nouvelle approche de la gestion de versions d'objets. Des règles pour

les autorisations basées sur le contenu des objets seront aussi présentées basées sur l'utilisation de vue, entité permettant d'introduire des conditions que les objets doivent vérifier afin d'être accessibles. Ce modèle pourra également supporter les autorisations basées sur les méthodes.

Pour ce qui est de la deuxième approche, des propositions de schémas compatibles avec les modèles d'objets à unique niveau de sécurité et plusieurs niveaux ont vu le jour. Par exemple considérons une classe student. Si l'on veut que la variable GAP de STUDENT soit secrète, créons une classe STUDENT-GAP ayant un niveau de sécurité SECRET, sous classe de la classe STUDENT.

Ou prenons le cas d'une classe EMPLOYEE, si le niveau de sécurité de la variable d'instance Adresse dépend du contenu de la variable profession, d'employé, il faut créer deux classes S-Adress et U-Adress sous classes d'employé représentant respectivement un employé ayant une adresse secrète et un employé non classifié.

Les problèmes de cette approche sont qu'en changeant la valeur d'une variable d'instance dynamiquement, le schéma est modifié. De plus certaines instances peuvent être à des niveaux de sécurité plus élevés que celui de leur classe ce qui pourrait rendre inaccessibles certains de leurs éléments. Puis les informations sont répétées à chaque niveau de sécurité ce qui constitue un problème de consistance.

Bertino et Jajodia proposaient également un modèle basé sur les composite objects. Au lieu de répéter l'information d'un niveau de sécurité bas dans les objets à haut niveau de sécurité, une référence à l'objet secret est introduite dans l'objet non classifié. Le problème ici est que les propriétés d'une entité à plusieurs niveaux de sécurité sont stockées dans plusieurs objets. L'entité nécessite donc l'accès à beaucoup d'objets.

Toujours en liaison avec la deuxième approche sera proposée une nouvelle démarche pour modéliser les entités à plusieurs niveaux de sécurité au travers de single-level objects grâce à des modèles de vues ou *views*. Les *views* sont pratiques car :

- Vu qu'on peut les considérer comme des sous-classes ou super classes virtuelles d'autres classes mères, on peut les modifier dynamiquement en gardant leur héritage intact.
- On peut les paramétrer arbitrairement sur plusieurs groupes de classes et d'autres vues avec différents niveaux de sécurité, donnant ainsi naissance à des multi–level views.
- On peut voir les vues comme des contraintes qui relient des données à d'autres données. On peut donc les utiliser pour retirer l'accès à certaines données pour parer les risques d'inférence et d'agrégation.
- Les views sont définies indépendamment des données qu'elles relient. De cette façon, elles peuvent être modifiées sans que toutes leurs données soient classifiées à nouveau.

Dans cette synthèse, le concept de *view* est utilisé pour modéliser une politique de sécurité à plusieurs niveaux. Les deux modèles seront dans un premier lieu présentés. Puis nous nous pencherons sur des concepts clés qu'ils partagent avant de conclure.

INTRODUCTION AUX MODELES

Le premier modèle

Quelques définitions

Pour être à même de comprendre le premier modèle d'autorisation proposé, il faut d'abord comprendre certaines notions des modèles de données orientés objet. Un modèle de référence sera ici présenté et le reste aura ce modèle comme inspiration. Ce modèle n'est pas un nouveau car il a des spécificités que l'on retrouve dans beaucoup d'autres modèles de la sorte, mais il a la particularité d'introduire les notions d'objets composite et de versions d'objets. Par la suite sera aussi présenté un langage de contraintes qui sera utilisé pour expliciter les règles d'autorisations dépendant de contenu concernant les objets et les sujets eux-mêmes.

La référence au modèle de données orientée objet

On retrouve dans ce modèle les concepts de base du langage de programmation orienté objet, c'est-à-dire les classes, leurs attributs, leurs méthodes, leurs objets et les principes de hiérarchie.

Cependant une notation spécifique sera utilisée :
Ai = ai: Ci représente la définition d'un attribut a une valeur ai étant le nom de l'attribut et Ci le nom de la classe de référence
Ai=ai :set-of Ci est la définition d'un attribut à plusieurs valeurs
M(In 1, In2, ..., Inn)---> Out est la définition d'une méthode avec M le nom d la méthode, Ini ($1 \leq i \leq n$) sont des paramètres pouvant être des objets et Out un paramètre de sortie. Les classes sont définies par leurs attributs, méthodes et superclasses, la classe TOP_CLASS étant la superclasse de base ou sans superclasse.

Les accessor methods aussi sont fournis par le système, les méthodes read et write, méthodes qui seront utilisées dans le modèle d'autorisation. La notation O.Ai représentera O.Read(Ai). D'autres méthodes comme Create, Delete, Define-class, Delete-class, Insert, Remove sont aussi fournies par le système.

La notion de composite object

Un objet composite est composé de références d'autres objets. On peut affiner les composite objects avec les spécificités shared|exclusive et dependent|independent. Shared signifie qu'un composant peut appartenir à plusieurs composite object, exclusive respectivement à un seul composite object. Puis l'existence d'un objet composant peut

dépendre de celle de son composite object ou pas. 4 possibilités sont donc introduites. La notion d'objet composite est intéressante d'un point de vue de performance car un composite object peut être utilisé comme une unité d'autorisation.

La notion de versioned object

Un *versioned object* correspond à une hiérarchie d'objet ou *version hierarchy.* Chaque objet de la hiérarchie dérive d'un autre en changeant un ou plusieurs attributs du premier. De surcroît, ces objets possèdent leur propre OID ou identifiant. Deux types de références sont désignés dans une hiérarchie de versions. La *dynamic reference* est la référence de l'objet qui se trouve à la base de la hiérarchie, objet plus communément appelé le *generic object*. La référence *static* correspond à un objet spécifique dans la hiérarchie d'objets. Certaines versions peuvent être modifiées ou supprimées par l'utilisateur, dans ce cas elles sont appelées *stable,* dont le *generic object* est en général la version la plus *stable*. Dans le cas contraire elles sont des versions de coupure ou dénotées *transient.*

Le langage de contraintes

1) Syntaxe

Le langage de contraintes est noté L et est un langage de premier ordre. Sa syntaxe se compose de variables, constantes, symboles de fonctions. Mais encore on y compte des prédicats dont l'arité, nombre d'arguments, est strictement positive. Un prédicat est une fonction mettant en œuvre un ou plusieurs arguments, utilisée pour déclarer quelque chose concernant des objets, et dont le résultat est vrai ou faux [4]. Les formules et leurs termes sont construits autour des symboles arithmétiques les plus courants.

Pour être utilisé dans le modèle d'autorisation, des extensions sont ajoutées au langage:

- le prédicat binaire *class* de la forme *class(t,τ)* pour un terme t et une classe τ.
- 3 prédicats ternaires : *ATT(c,a,d)* signifiant qu'une classe de nom *c* a un attribut de nom *a* et de type *d*, *ARITY(c,a,n)* indiquant l'arité *n* de l'attribut *a* de la classe *c* et *VAL(o,a,x)* signifiant que l'attribut *a* de l'instance *o* compte x parmi sa/son ensemble de valeurs.

Exemples :

ATT(document, authorlist, employee) : la classe document a un attribut de nom authorlist et de type employee.
VAL(c1,title,"an approach to authorization modeling") : l'objet c1 a un attribut de nom title et de valeur "an approach to authorization modeling".
Les attributs à unique ou égales à *n* sont respectivement *ATT1* ou *ATTn*.

2) Le chemin ou path

Le chemin ou path est défini par : si *x* est une variable ou constante et *a* le nom d'un de ses attributs, *x.a* est chemin de base, respectivement si p est un chemin et a le nom d'un attribut alors p.a est un chemin.

Un chemin est *single-valued* si tous ses attributs ont une arité égale 1 sinon il est dit *set-valed*.

Les expressions de chemin interviennent dans les relations d'égalité et d'appartenance :

x.a = v ⇔ v = x.a ⇔ VAL(x, a, v) pour des attributs single-valued et
$v \in x.a \Leftrightarrow VAL(x, a, v)$ pour des attributs set-valued avec v constante.

De même pour un chemin set-valued,

$v \in p.a \Leftrightarrow \exists\ c\ (c \in VAL(c, a, v))$ *p* étant un chemin et *c* une nouvelle variable
$v \in p.a \Leftrightarrow VAL\ (p, a, v)$ quand *p* est un atome

Exemple : l'expression $y \in x.authorlist.name$ est équivalente à :

$\forall\ c\ (\ VAL(x, authorlist, c) \wedge VAL(c, name, y)\ )$ toute variable *c* de l'attribut *authorlist* de x aura un attribut *name* de valeur *y*.

Écrivons sous L une contrainte qui expliciterai que si un employé est sur la liste d'auteurs d'un document alors son manager est dans liste aussi :

$\forall$ e: Employee, $\forall$ d: Document(e $\in$ d. authorlist => e. manager $\in$ d. authorlist))

Le modèle d'autorisation

Ce modèle d'autorisation établit des règles aussi bien dépendantes qu'indépendantes du contenu des objets dont elles régulent l'accès. Dans le premier cas, il est requis d'accéder aux objets pour appliquer la règle d'autorisation, dans le 2ème il n'est pas requis de faire cela.

Vue d'ensemble du modèle d'autorisation

Soit *S* un ensemble de sujets, *O* un ensemble d'objet et *A* un ensemble de type d'autorisations (lire, écrire..).

Une autorisation explicite est définie par le prédicat b_auth(s,o,a) où $s \in S$, $o \in O$, $a \in A$.

Exemple : b_auth(u_i,d[1],Read) signifie qu'un utilisateur u peut lire le document d[1].

L'*Explicit Authorization Base* est l'ensemble de toutes les autorisations explicites ou EAB. Le prédicat auth(s,o,a) détermine si une autorisation (s,o,a) est vraie ou fausse.

La mise en place d'autorisations repose donc sur un jeu de logique, en effet :
$\forall$ S: S, $\forall$ o: O, $\forall$ a: A (b_auth(s,o,a)=>auth(s,o,a)), un sujet a une autorisation de type a sur l'objet o si une autorisation explicite existe.

auth sera utilisé aussi bien pour les autorisations explicites qu'implicites, et si on ne peut pas prouver que cette fonction est vraie, elle retournera faux.

Un point de vue pour gérer les autorisations serait de lister toutes les autorisations explicites possibles, mais il n'est pas judicieux et nécessaire de le faire car des autorisations dites implicites peuvent découler d'autorisations explicites. Au lieu de mettre les règles implicites dans l'EAB on peut établir des règles qui autorisent les règles explicites à dériver des règles implicites. L'ensemble de ces règles est groupé sous le *Intensional Authorization Base* (IAB). L'Authorization Base (AB) est donc composé de l'*EAB* et de l'*IAB*.

En outre certaines autorisations explicites peuvent dépendre du contenu des objets qu'elles régulent.

Le prédicat utilisé pour ces règles est c_auth().

Par exemple un prédicat « un employé peut lire un document seulement s'il appartient à la liste des auteurs de ce document» s'écrit :

$\forall$ e: Employee, $\forall$ d: O(e $\in$ d.authorlist => c_auth(Document, e, d, Read))
par analogie avec la règle explicite, (c_auth(C,s,o,a) $\wedge$ o$\in$ C)=>auth(s,o,a))
ce qui implique que *e $\in$ d.authorlist => auth(e,d,Read))*

Dans les règles de type c_auth(), l'autorisation est liée à la classe de l'objet ce qui facilite le concept d'héritage. Mais l'autorisation peut également dépendre d'une valeur d'une variable d'instance spécifique. Par exemple, si on veut autoriser l'accès à un document d[3] que lorsque il est sous statut « released », on peut écrire :

$\forall$ e:Employee $\forall$ o:O (o=d[3](o.status='released' => c_auth(Document,e,o,Read)).

Les 2 types de règles *content dependent* et *independent* peuvent interférer entre elles. En particulier quand une règle *independent* et une règle *content-dependent* s'exercent sur un même objet, il suffit que leur sujet vérifie l'une des deux pour avoir l'autorisation sur l'objet.

De plus, un utilisateur qui reçoit une autorisation explicite et indépendante pour lire un objet par un autre utilisateur sans vérifier une autre autorisation explicite dépendante pour

ce même objet (appartenir à liste d'auteur pour lire l'objet) est au final autorisé à accéder à l'objet.

La politique de base du modèle est qu'un utilisateur peut exécuter une action sur un objet s'il possède une autorisation *content-dependent* ou *content-independent* sur celui-ci.

Les règles d'implication pour les sujets d'autorisations

Les utilisateurs peuvent être regroupés sous le nom de rôles. Cela permet d'assigner des autorisations à des groupes de personnes plutôt qu'au cas par cas. Tous les rôles sont organisés dans le *role graph* R.

Une des propriétés les plus basiques de ce graphe est que si un utilisateur appartient à un rôle, il appartient à tous les super rôles de ce rôle :

On a 2 rôles r_i et r_k tel que $r_i \leq r_k$, $\forall$ *s: S(s* $\in$ r_i*=>s* $\in$ r_k*)* (cette règle sera notée Ir).

Donc, si des autorisations sont valables sur r_i alors elles sont valables sur r_k.

$\forall$ *s: S(s* $\in$ *ri* $\wedge$ $r_i \leq r_k$ $\wedge$ *auth(r_k, o, a) => auth(s, o, a))*

On peut également utiliser ce concept de rôles pour les règles *content-dependent*, par exemple avec l'exemple de l'employé qui peut lire un document s'il appartient à la liste d'auteurs, le manager étant un super rôle d'employé il hérite de cette autorisation.

$\forall$ *e: Employee,* $\forall$ *d: Document(e* $\in$ *d.authorlist =>auth(e,d,Read))*

Par Ir on déduit que

$\forall$ m: Manager, $\forall$ d: Document(m $\in$d.authorlist => auth(m,d,Read))

Le manager obtient ici une autorisation implicite content-dependent. Donc les autorisations d'un sous rôles sont valables pour tous ses super rôles.

De plus le manager peut obtenir l'autorisation de lire le document s'il appartient à la liste des managers du projet du document (m = d.project.manage). Pour obtenir l'autorisation de lire un document, il doit remplir une condition dépendante de la disjonction de ces 2 règles.

Les rôles peuvent être modélisés par les même classes utilisées dans le modèle d'autorisation, la différence est que ces classes représentant des rôles sont dérivées de la classe de base user.

Les règles d'implication pour les différents types d'autorisation

On peut également déduire des autorisations grâce à des types d'autorisation de base. Posséder le droit d'écrire dans un objet devrait impliquer d'avoir le droit de le lire aussi par exemple. Grâce à ces règles d'autorisations sur certains objets, l'utilisateur peut donc bénéficier de toutes les autorisations qui découlent de celle de base.

Il faut donc définir quels sont les différents types d'autorisation possibles dans un premier temps.

Table 1
Authorization types

Authorization type		Actions allowed
Datebase	Read	Read the database definition
	Read-all	Read all objects within the database
	Write-all	Modify all objects within the database
	Create	Create a class
Class	Read	Read class definition
	Write	Modify class definition
	Delete	Delete a class
	Read-all	Read all instances of the class
	Write-all	Update all instances of the class
	Read(Ai)-all	Read attribute Ai for all instances of the class
	Write(Ai)-all	Update attribute Ai for all instances of the class
	Create	Create new instances of the class
Instance	Read	Read all attributes of the instance
	Write	Modify all attributes of the instance
	Delete	Delete the instance
	Read(Ai)	Read attribute Ai of the instance
	Write (Ai)	Modify attribute Ai of the instance; also insert, and remove if Ai is multivalued

Chaque type d'autorisation spécifie plusieurs méthodes définies par le système. Certaines ne correspondent pas à des méthodes définies par le system mais sont plutôt des raccourcis d'ensemble d'autorisations. Les méthodes telles que *Read-all* ou *Read(Ai)-all* dites *parametric* accordent le droit de lire toutes les instances d'une classe ou tous les attributs de toutes les instances de cette classe. Par exemple, *b_auth(Employee,Document,Read(abstract)-all).* Les employées peuvent lire les abstracts de tous les documents.

Définir des règles d'implication nécessite la définition de plusieurs termes : l'ensemble des objets O est divisé en 3 sous-catégories. La *Database*, réunissant tous les objets de la base de données, puis l'ensemble de ses classes *Class* et des instances de ces classes *Instance*.

Certaines sont communes aux Class et Instance :

I_{o1} $\forall$ s S, $\forall$ Oi: Class U Instance (auth(s, Oi, Write) => auth(s, Oi, Read))
I_{o2} $\forall$ s: S, $\forall$ Oi: Class U Instance (auth(s, Oi, Delete) => auth(s, Oi, Read))

D'autres règles d'implication s'ajoutent pour les classes :

I_{c1} $\forall$ s: S, $\forall$ Ci: Class (auth(s, Ci, Read-all)=>auth(s, Ci, Read))

Ici lire les instances d'une classe implique de pouvoir lire la classe de définition ce qui paraît logique.

I_{c4} $\forall$ s: S, $\forall$ Ci: Class, $\forall$ Ai: Att-set(Ci) (auth(s, Ci, Write-all) =>auth(s, Ci,Write-all(Ai)))
I_{c6} $\forall$ s: S, $\forall$ Ci: Class (auth(s, Ci, Create) => auth(s, Ci, Read))

L'autorisation Read est donc impliquée par toutes les autres autorisations.

De même pour les instances,

I_{11} $\forall$ s: S, $\forall$ I_i: Instance, $\forall$ Ai: Att-set(I_i) (auth(s, I_i, Write) => auth(s, I_i, Write(Ai)))

Par contre il existe une exception : si un utilisateur reçoit l'autorisation de lire les attributs d'un objet et que ce même objet est une référence sur un autre objet, l'utilisateur ne peut lire que l'OID de l'objet référencé.

Pour les autorisations de type *database*, l'utilisateur possédant une autorisation Read peut consulter le répertoire de classes ainsi que la hiérarchie des différentes bases de données. Cependant l'utilisateur ne peut voir la définition des classes. Seule la fonction *Read-all* de *database* permet de lire la définition des classes et de leurs instances.

Les autorisations de type *content-dependent* suivent le même schéma que celui qui vient d'être présenté. Cependant les conditions doivent être des *class-attributes* partagés par les classes des database en question. Ces *class attributes* sont des attributs caractérisant les classes en tant qu'objet et ne sont pas hérités des instances d'autres classes. Un *class-attribute* pourrait être la date de création de la classe.

La différence avec l'ancien modèle d'autorisation est qu'un objet setofinstances devait être créé pour les règles d'autorisation de type Read-all. Ici les règles d'implication font le même travail sans overloader les autorisations de type. De plus contrairement à précédemment, ici l'autorisation de modifier une définition de classe n'implique pas de modifier le contenu de ses instances mais juste leur structure.

Les règles d'implication pour les autorisations sur les d'objets

Cette 3ème dimension de règles d'implication considère les objets d'autorisation. On parle *d'authorization granularity hierarchy* et *d'object authorization inheritance*.

Chaque nœud de la *authorization granularity hierarchy* représente un objet de la base de donnée et chaque nœud a des autorisations le concernant et des autorisations concernant certains des objets à des niveaux plus bas.

Par exemple une classe peut être ciblée par des autorisations de lecture/écriture de sa définition, mais également de lecture/écriture de ses instances. On factorise ici les autorisations sur les instances de la classe. Dans une granularité d'objet, on peut donc propager les autorisations d'un objet dans les objets sous-jacents.

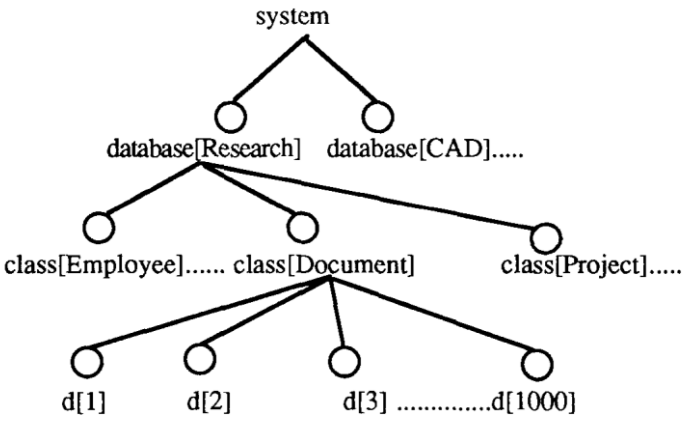


Fig. 3. Example of object authorization granularity hierarchy.

Pour définir les règles d'implication de ce type, on considère 2 objets Oi $\in$ Oj et Oi $\in$ Oj signifie que Oi est un descendant immédiat de Oj.

Elles s'effectuent à deux niveaux de granularité : de database vers classe et de classe vers instance.

- De database vers classe

lDC1 $\forall$ s: S, $\forall$ Di: Database, $\forall$ Ci: Class ((auth(s, Di, Read-all) $\wedge$ Ci $\in$ Di)), auth(s, Ci, Read-all))

- De classe vers instance

Un utilisateur ayant une autorisation read-all sur une classe peut donc lire les instances de cette classe, de même pour Write all et le mécanisme opère pour les attributs d'une définition de classe et les attributs des instances de cette classe. Cependant ici on ne considère que les instances ayant les mêmes attributs que ceux de la classe mère. Si des attributs supplémentaires sont ajoutés, alors l'autorisation n'est plus valide et il faut faire une autorisation au niveau de l'instance.

Toutes les autorisations précédentes suivaient le modèle top-bottom. Quelques règles implicites peuvent suivre le modèle bottom-top.

IC15 $\forall$ s: S, $\forall$ Ci: Class, $\forall$ Ii: Instance, $\forall$ Ai: Att-set(Ci) ((auth(s, Ii, Read(Ai)) $\wedge$ Ii $\in$ Ci) => auth(s, Ci, Read))

Un utilisateur pouvant lire un attribut d'une instance peut lire la définition de la classe correspondante.

De plus les règles *content-dependent* peuvent s'appliquer à ces règles implicites :

$\forall$ m: Manager, $\forall$ d: Document (m $\in$ d. authorlist => auth (m, d, Read))
Implique que
$\forall$ m: Manager, $\forall$ d: Document (m $\in$ d.authorlist => auth(m, Document, Read))

Complétude des règles d'autorisation

Globalement, les règles d'autorisation sur les objets couvrent complètement toutes les formes de dérivation possible. En effet elles respectent 4 principes :

- Si un utilisateur est autorisé à écrire ou supprimer une entité, il peut implicitement la lire
- Tout type d'autorisation (lire, écrire, supprimer, créer) implique l'autorisation de lire
- Si un utilisateur peut lire ou écrire dans une entité, il peut donc lire ou écrire tout ce qui compose cette entité
- Si un utilisateur a une autorisation read all ou write all sur une entité e, alors il a une autorisation read ou write sur chaque sous entité de e en respectant la hiérarchie granulaire d'entité de e.

Les autorisations pour les méthodes définies par les utilisateurs

Jusqu'à maintenant on ne considérait que les *system-defined methods*. On peut également se pencher sur les *user-defined methods*. 2 approches sont considérées. La première est d'évaluer toutes les autorisations nécessaires pour l'exécution de la méthode. Un utilisateur voulant copier un document doit être autorisé à lire le document et créer une instance de la classe document. Cependant certaines méthodes ne nécessitent pas d'avoir de telles autorisations pour opérer. La 2ème approche, *l'execution under protection mode* est de vérifier les autorisations de la méthode pour un autre utilisateur que celui qui l'exécute. C'est souvent l'utilisateur qui possède la classe et qui autorise notre utilisateur à l'exécuter.

Les autorisations sur les méthodes *user-defined* sont de cette forme :

b_auth(u, u', m, o, p) où u et ù sont les utilisateurs, m est le nom de la méthode *user-defined*, o un objet et p une balise qui prend pour valeur soit True ou False. u reçoit le

privilège d'exécuter m sur o par ù. Si p est à True, tous les accès de la méthode sont vérifiés auprès de u, ce qui correspond au mode d'*execution under protection*.

Exemple : (Black, Smith, ReviewStatus, Document[i], True)
Black autorise Smith à exécuter sous le mode protection la méthode ReviewStatus sur le Document[i].

On peut également appliquer des règles implicites pour ces autorisations sur méthodes user-defined : les autorisations de méthodes sur une classe s'appliquent également sur les instances de cette classe.

Le deuxième modèle

Les bases de données au niveau de sécurité multiple

Les bases de données contiennent des informations à différents niveaux de sécurité, ce qui implique que certains utilisateurs n'ont pas accès à certaines données. On attribue donc à chaque entité un paramètre de sécurité à chaque entité : un *clearance level* ou degré d'autorisation aux entités actives (sujets) et un *security level* ou niveau de sécurité aux entités passives (objets). Ces derniers paramètres ne peuvent être modifiés que par des responsables de sécurité.

Les niveaux de sécurité et d'autorisations sont organisés par ordre croissant. Le degré TOP_SECRET est supérieur au degré SECRET lui-même supérieur au degré CONFIDENTIAL. On peut également rajouter des compartiments aux degrés de sécurité : un degré de sécurité [SECRET, ASIACRYPT] donne une autorisation aux un utilisateur de type SECRET appartenant au compartiment ASIACRYPT, donc pas tous les utilisateurs de type SECRET ont accès aux données de ce degré de sécurité.

L'encapsulation des données combinées au label de sécurité sont suffisant pour protéger les objets concernés mais par le flux d'information à proprement parler. Le modèle le plus connu qui protège le flux d'information est le système Bell-Lapadula.

Ce système se base sur la propriété de type *simple security* et des propriétés multiples dites *-property. La première propriété donne accès à la lecture de l'objet par un utilisateur si celui-ci domine le degré de sécurité de l'objet (Lsubject ≤Lobject). Le second concerne l'accès à l'écriture de l'objet pour des conditions inverses.

Le modèle de Vues

Le modèle de sécurité est basé sur le modèle de View proposé par Bertino, modèle sans lien avec la sécurité.

Smith avait évalué 3 dimensions de protection des données :

1. La donnée peut être classifiée
2. L'existence de la donnée peut-être classifiée
3. La raison pour laquelle la donnée est classifiée peut-être classifiée

L'*acces-view* est proposée pour couvrir les 2 premières dimensions et les *security constraints* couvrent le 3ème. Les méta-classes ACCESS VIEW renforcent donc le contrôle de l'accès aux données et les méta-classes *security constraints* contrôlent le niveau de sécurité des *Access-views* elles-mêmes.

Une instance d'ACCESS VIEW présente cette structure :

access-view av-name [parameters]
[**properties** list_of_property_names_and_security_levels]
[**view-query**]
[**additional-properties** list_of_properties_and_security_levels]
[**methods** list of method_specifications]
[**superviews** list of views]
[**OID** true or false]
[**identity-from** list of class_names]
[**av_level** view_security_level]
[**av_range** upper_and_lower security_level]
[**mac-constraint** set-of < SECURITY-CONSTRAINTS > mac_names]
av_name : nom de l'access view, il se doit d'être unique, les valeurs des paramètres sont des objets assignés dynamiquement lors de l'évaluation de la view.
view query : définit l'access-view par une demande ou *query* ou sur un ensemble arbitraire de classes et de classes dérivées ou view. Ce sont les *base classes.*
De plus on peut ajouter des conditions *content-based*, incluant le contenu d'une des propriétés devant être vérifiée dans la query.
properties : l'ensemble de variables d'instance et leur niveau de sécurité pouvant dériver des *base classes*
additional properties : ou pouvant-être ajoutées
methods : la view peut aussi être associée à des méthodes héritées des base classes ou introduites ici
OID : indique si des identifiants persistants des instances de view doivent être retournés pour lors de l'exécution de la View. Si OID est à true ils sont retournés après une query sur une view. Si des identifiants sont effectivement retournés, l'utilisateur peut envoyer des messages aux instances et les utiliser les comme des classes
identity-from : lorsque pour les views dont la view query joint d'autres views, on y inscrit le sous ensemble de base classes qui détermine l'identité de la view

av_range : impose des restrictions sur la gamme de niveaux de sécurité des base classes ou views et propriétés

av_level : indique le niveau de sécurité du nom de la view. Quand la av range n'est pas indiquée, la view contient les single-level properties et classes de base indiquées par leurs av-level

mac constraints : contraintes de contrôle d'accès ou ensemble de règles de sécurité que la view doit satisfaire. Chaque élément ici est une instance de la méta classe SECURITY CONSTRAINTS.

Exemple :

Reprenons l'exemple de la classe Student dont l'instance de la variable GAP est classifiée secrète et les autres non classifiées.

On peut représenter la classe Student en deux view : U_STUDENT dont le niveau de sécurité est non classifié et S_STUDENT dont le niveau de sécurité est classifié avec la variable secrète GAP et la superview U_STUDENT.

access-view U_Student
(view query)select S.idno, S.name, S.age, S.status, S.address, S.spouse, S.sex, S.subject,
S.Start-Date, S.Graduate-Date, S.takes
from S:STUDENT
  av level UNCLASSIFIED
access-view S_Student
properties [Greatest_Point, SECRET]
(view query)select S.GAP from S:STUDENT
superview U_Student
av_level SECRET
av_range [UNCLASSIFIED, SECRET]

L'existence d'une U_STUDENT view suggère au student non classifié qu'il y a une view secrète d'où l'utilisation de la méta classe SECURITY-CONSTRAINTS. Ses instances suivent ce format :

security-constraint sc-name [parameters]
[predicates] [property-name.level = property_security_level]
[predicates] [view-name.level = view_security_level]
[sc_level sc security level]
sc-name : nom unique de la contrainte de sécurité différent de noms de views ou autres classe ou security constraints
sc-level : indique le niveau de sécurité de la contrainte de sécurité, peut aussi indiquer celle de la View et de ses propriétés

Les contraintes de sécurité peuvent être *simple*, *content* ou *aggregate*.

Simple : classifie une propriété entière de view ou la view elle-même.
content : permettent de classifier les données au niveau de sécurité d'objet ou de propriété en utilisant un prédicat basé sur la valeur de certains objets ou propriétés.
aggregate : classifient une collection de valeur de propriétés ou une relation à un niveau de sécurité élevé.

Le modèle de vues à niveaux de sécurité multiple

Les *multi-level views* couvrent les objets à niveau unique de sécurité. Il faut rappeler qu'ici le principal but est de renforcer la politique de sécurité mandataire – cette politique assure que dans une base de donnée, aucun sujet ne peut obtenir une information sans autorisation et aucun mécanisme n'existe où un sujet pourrait communiquer de l'information à un autre sujet non autorisé à y avoir accès.

On considère que les entités à classifier sont tout type d'objet (access views, des contraintes de sécurité, des classes, des objets, méthodes..) ayant un niveau de sécurité. Les méthodes suivantes sont disponibles pour se renseigner sur les informations sécuritaires d'un élément e.

Level (e) - montre le niveau de sécurité de l'entité e.
Lower (e)-montre la gamme de niveau inférieure d'entité e.
Upper(e) - montre la gamme de niveau supérieure d'entité e.

Avec la plus petite des gammes de niveau supérieure de e notée LUB et la plus grande des gammes inférieures de e notée GLB.

Les règles sur les views

On peut considérer une entité selon un approche top down ou bottom up. Top down signifie qu'une grosse classe de base et utilisée pour créer d'autres classes spécialisées. Bottom up signifie que beaucoup de classes ou views sont combinées entre elles pour former une grosse classe par un processus de généralisation. Cette dernière approche peut permettre de rassembler beaucoup d'information au sein d'une view, cette dernière devant donc avoir une classification supérieure à celle de ses composants. Les règles suivantes seront basées dessus.

(**propriété de la view**)
Règle 1:La gamme de niveau d'une view doit contenir celle de la view
v.av-level $\geq$ LUB{v1.av-level, v2.av-level..... vn.av-level}
Pour un souci de pratique l'on peut considérer les propriétés et méthodes d'une view comme ses *facettes*.

(Propriétés des facettes d'une view)

Règle 2 si x est une facette de v et est dérivé d'une view de base w, le niveau de sécurité de x doit dominer celui de v

Level(v.x) ≥ v.av-level

Règle 3 : La gamme de sécurité d'une view doit contenir tous les niveaux de sécurité des facettes de v

GLB{Level(v.x I), Level(v.x2)..... Level(v.xn)} ≥Lower(v),

et LUB{Level(v.xj), Level(v.x2)..... Level(v.x)} ≤Upper(v).

(Propriété de l'utilisateur)

Un utilisateur doit d'abord avoir accès au nom de la view s'il veut avoir accès à son contenu. Il doit donc avoir un degré d'autorisation qui doit dominer le niveau de sécurité de la view.

Règle 4 : Un utilisateur peut accéder à une view si, quand la view est une single level view, il domine son niveau de sécurité et quand la view est multi level, il domine son plus petit niveau. Pour accéder à une facette d'une view, il doit ensuite dominer le niveau de sécurité de la facette. On considère ici la read up approche du modèle Bell-LaPadula.

(Propriété des instances d'une view)

Règle 5 : Le niveau de sécurité d'une view doit dominer le LUB de ses objets.

v.av-level ≥LUB{Level(o1), Level(o2)..... Level(on)}.

(**Propriétés des contraintes de sécurité**)

Règle 6 : Pour des facettes x1…. xn et s1… sn dont les niveaux de sécurité sont contenus dans la contrainte de sécurité de sc :

- les niveaux de sécurité des facettes contenus dans la sécurité de contrainte doivent dominer ceux de leur facettes xi si≥Level(xi)
- le niveau de sécurité de la *security-constraint* est le LUB de tous les niveaux de sécurité des facettes contenues dans la sécurité de contraintes

Règle 7 un utilisateur peut accéder à une sécurité de contrainte s'il domine son niveau de sécurité

Règle 8 : si une view v hérite d'une w, alors son niveau de sécurité domine celui de la view w

Règle 9 : si une view v hérite de la facette x d'autres view vi, alors le niveau de sécurité de x dans v domine le LUB de x dans les autres vi

Règle 10 : si v est un composé de view, alors v domine le LUB de ses autres *view composites*
v.av-level≥LUB{vl.av-level, v2.av-level..... v,.av-level}

Règle de dérivation

Un objet dont le niveau de sécurité est dominé par une single level view peut-être dérivé comme instance de cette view même chose si le niveau de sécurité d'une valeur de ses propriétés est dominé par le niveau de sécurité de la propriété correspondante d'une multi-level view.

Mises à jour

Contrairement aux bases de données relationnelles simples, les bases de données orientées objet permettent de garder la trace des objets de base, ce qui rend plus facile les insertions d'objet notamment dérivés.

Pour ce qui est du niveau de sécurité, il dépendra de ceux ou celui de l'objet et des contraintes de sécurité de la view correspondante. Un objet à plusieurs niveaux de sécurité doit être divisé en single level object lors de son insertion en base de données en fonction de chaque propriété de la view.

CONCEPTS CLES

L'héritage

Dans le premier modèle il est question de savoir si les autorisations appliquées à une classe devraient être automatiquement héritées par ses sous classes. Un tel modèle d'autorisations faciliterait la tâche si on ne voulait pas remettre en place des autorisations sur une sous classe, mais il poserait problème si l'on ne le souhaite pas ou si on souhaite renforcer la protection des sous classes. Les autorisations dans le modèle de ce document ne seront donc pas automatiquement héritées mais l'utilisateur aura le choix de le faire.

Une relation d'héritage autorisation sera représentée par $\leq^A$. C $\leq^A$ C' signifie que la classe C hérite automatiquement de toutes les autorisations de la classe C' avec C sous-classe de C'. L'utilisateur peut entrer une telle autorisation grâce à l'interface prévue à cet effet.

L'on doit également distinguer les relations d'héritage pour les règles *content-independent* noté $\leq^B$ et les relations pour les règles *content-dependent* noté $\leq^C$. En effet certaines règles d'une classe ne pourrait être valables pour une sous classe à cause de l'overriding (même méthodes mais dans sous classe comportement différent) et de l'héritage multiple. Avec ces concepts, un attribut d'une sous classe pourrait avoir une autre définition que dans la super classe. Les règles *content-dependent* pourrait donc être invalides.

On devrait donc redéfinir les règles d'autorisation *content-dependent* dans une sous classe.

La relation $\leq^A$ regroupe les deux autres. On peut combiner l'héritage de règles avec les règles implicites citées plus haut.

Exemple :

b_auth(Manager, Document, Read-all))
$\forall$ e: Employee, $\forall$ o: Object(e $\in$ o.authorlist => c_auth(Document, e,o,Read)) (autorisation de type implicite)
or Memo $\leq^A$ Document
donc auth(Manager, Memo, Read-all))
$\forall$ e: Employee, $\forall$ o: Object(e $\in$ o.authorlist => c_auth(Memo, e,o,Read))

L'autorisation de créer une sous classe fait intervenir 2 autorisations : il faut avoir l'autorisation de créer une classe et l'autorisation de lire la définition de la classe mère.

Dans le deuxième modèle, certaines View peuvent hériter d'autres views lors de leur création. Cette opération se fait en indiquant les views de base dans superviews. La view hérite de toutes les variables d'instances, méthodes et contraintes des views de base.

Reprenons l'exemple de la classe Student dont l'instance de la variable GAP est classifiée secrète et les autres non classifiées.

U_STUDENT, libellée UNCLASSIFIED, est une view à niveau de sécurité unique contenant toutes les variables d'instance de la base classe STUDENT sauf pour GAP. S_STUDENT héritant de U_STUDENT est une *multi level view*, en plus de sa variable secrète Greatest_Point elle hérite de toutes les propriétés de U_STUDENT.

La composition

Dans le premier modèle on considère que les composite objects constituent une unité d'autorisation. Une autorisation sur un objet composite devrait s'appliquer à tous les objets qui le composent. On utilise ici la notation I pour un objet composite et comp-set(I) pour toutes les instances qui y sont référencées.

Dans la figure suivante par exemple on a deux documents d[1] et d[15] partageant une section s[14].

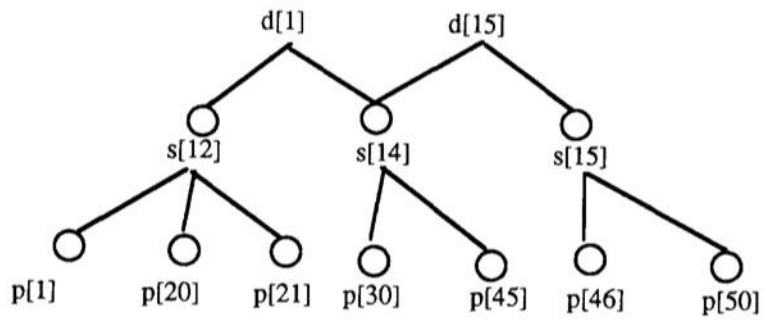


Fig. 5. Composite object examples.

Peu de règles d'autorisation et de types d'autorisation sont nécessaires pour couvrir l'ensemble des règles d'autorisation à appliquer sur les objets composites.

$I_{Comp\ 1}$ $\forall$ s: S, $\forall$ Ii, $\forall$ Ij: Instance ((auth(s, Ii, Read-composite) $\wedge$ Ij $\in$ comp-set(Ii)) => auth(s, Ij, Read-composite))
$I_{Comp\ 2}$ $\forall$ s: S, $\forall$ Ii , $\forall$ Ij: Instance ((auth(s, Ii, Write-composite) $\wedge$ Ij $\in$ comp-set(Ii)) => auth(s, Ij, Write-composite))

On peut appliquer ces règles récursivement et voir qu'elles se propagent dans les instances directes ou indirectes des composants de l'objet racine. On introduit également le type d'autorisation *read-Composite*. Pour une autorisation Read, on peut lire les attributs d'un objet sauf si l'attribut est une référence. Pour l'autorisation *Read-Composite*, on peut également lire la référence/attribut et les attributs de cette référence.

Par implication, *Read-Composite* permet de lire les attributs/références et surtout Write permet de lire comme indiqué dans les règles précédentes. Cependant tous les composants qui ne sont pas issus de la classe root ne reçoivent aucune autorisation.

Par exemple un autorisation Read-Composite sur d[1] permet de Read s[12], s[14, p[1], p[20], p[21], p[30], p[45] mais pas s[15] et ses composants.

En créant un objet, on obtient *Write Composite* que si on a Write sur tous ses composants ou qu'on a créé ces composants. Sinon reçoit une autorisation Write dessus. Une fonction UPGRADE permet de passer de Write à *Write-Composite*, mais elle ne peut être effectuée que par un utilisateur possédant cette fonction. Les règles énoncées dessus s'appliquent donc pour ce genre d'utilisateur. Si l'autorisation *Write-Composite* pour le possesseur d'un objet composite est downgraded, alors tous les utilisateurs qui ont l'autorisation *Write-Composite* dessus se voient déclassés aussi.

Nous travaillions à un niveau d'instance. On peut également travailler à un niveau de classe en introduisant les règles *Read/Write-Composite-All* dont le fonctionnement est le même que Read et Read-All.

I_{C16} $\forall$ s: S, $\forall$ Ci: Class, $\forall$ Ii: Instance ((auth(s, Ci, Read-composite-all) $\wedge$ Ii $\in$ Ci) auth(s, Ii, Read-composite)).

En ce qui concerne les règles d'autorisations *content-dependent*, leur propagation est freinée par le fait qu'un objet composant vient souvent d'une classe différente de l'objet composite. On ne peut là aussi que trouver des attributs communs tels que la date de création de l'objet. Ou on peut créer des conditions qui s'appliquent à l'objet composite et des conditions qui s'appliquent à ses composants :

$\forall$ e: Employee, $\forall$ s: Section, $\forall$ d: Document ((s $\in$ comp-set(d) $\wedge$ e $\in$ s.section_ authors $\wedge$ d.project, research_programme =' ESPRIT') => auth(e, s, Read- composite))

Un employé peut lire la section d'un document s'il est dans la liste d'auteurs et le document est lié au projet ''Esprit''.

Dans le deuxième modèle, certaines View peuvent être des composée d'autres *views*, dans ce cas on utilise la propriété *additional_properties* pour ajouter une *base class* ou ici *base view*. On appelle ces *views* les *composite views*.

Les versions

Les versions constituent également des unités d'autorisation. Mais elles ne sont considérées que dans le deuxième modèle. Un *versioned* objet peut avoir plusieurs variations d'objets ou *versions*. Il constitue donc une unité d'autorisation. On utilise ici la notation *I* pour un *versioned* objet et *version-set(I)* pour toutes les instances qui sont des variations de *I*, donc toutes les variations d'objet de *I*.

Les règles d'implications pour les variations d'objet sont :

I*Vers1* $\forall$ S: S, $\forall$ Ii, Ii: Instance ((auth(s, Ii, Read) $\wedge$ Ij $\in$ version-set(Ii)) => auth(s, Ij, Read))
I*Vers2* $\forall$ S: S, $\forall$ Ii, Ij: Instance ((auth(s, Ii, Write) $\wedge$ Ij $\in$ version-set(Ii)) => auth(s, Ij, Write))

On peut dériver ces règles à un niveau d'attribut également.

Une question centrale sur les autorisations concernant les versions d'objets concerne l'autorisation de modifier des hiérarchies de variations d'objets telle que la création d'une variation *transient* ou de changer l'objet *versioned*. On peut développer des autorisations Write sur l'objet versioned pour régler le problème au lieu d'agir au cas par cas. En effet l'utilisateur pourra effectuer toutes les actions d'administration du versioned object impliquées par ces autorisations.

Une question se pose lors de la création d'une nouvelle version d'un objet *versioned*. Si on représentait cette autorisation par l'autorisation Write, on donnerait également l'autorisation Write sur toutes les versions de l'objet *versioned* alors que créer une version est une fonction qui permet de ne pas modifier les versions déjà existante. Une solution

serait de créer un type d'autorisation noté *Create* sur les classes, même si on autoriserait l'utilisateur à créer d'autres hiérarchies de versions à partir d'une version d'un objet *versioned*.

Ici est présenté une autorisation de type Create pouvant s'appliquer sur une instance *I* appartenant à une hiérarchie de versions. Cette solution permet donc de limiter la prolifération de versions dans une base de données car *I* doit appartenir à une hiérarchie de versions et *I* doit être une version *stable*.

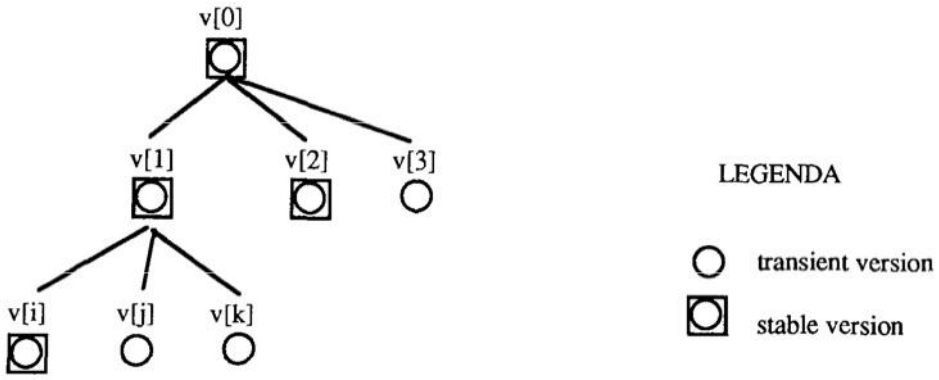


Fig. 6. Version hierarchy example.

En considérant cette hiérarchie de versions et les autorisations suivantes

auth(si, v[0], Create) et auth(sj, v[i], Create)

Le sujet si peut créer des versions de v[0], v[1], v[2], v[i] alors que le sujet sj ne peut en créer que de v[i].

Une règle implicite est que si un utilisateur peut créer une version d'une instance/classe, il peut donc la lire.

Au niveau des règles d'autorisation *content-dependent*, il y a beaucoup de possibilités étant donné que les attributs de l'objet *versioned* se retrouvent dans ses versions. On notera par ailleurs la condition Is_Stable permettant d'appliquer une autorisation aux versions stables d'un objet *versioned*.

L'administration des autorisations

Dans le premier modèle, l'approche repose sur *l'ownership approach* : un utilisateur étant propriétaire d'un objet, souvent le créateur de l'objet peut gérer les autorisations sur cet objet et accorder ce droit de gestion à d'autres utilisateurs ou le révoquer.

Dans les bases de données orientées objet, on devrait être propriétaire de l'instance d'une classe, instance qu'on a créée. Cela permettrait à des instances d'une classe d'avoir des propriétaires différents. En effet un utilisateur voulant être propriétaire d'une instance

d'une classe dont toutes les instances appartiennent à quelqu'un d'autre devrait dans ce cas créer une sous classe. Grâce à ce modèle, le propriétaire d'un objet peut également changer. C'est une vision de décentralisation des autorisations sur les instances d'une classe.

Seulement un autre problème intervient avec la granularité des classes supportée par le modèle : des droits tels que Write-all ou Read-all sont très puissants et on ne sait pas quels utilisateurs pourraient accorder de tels droits. Il faut donc introduire la possibilité pour les classes d'être administrées par une personne : une classe a un *class administrator* propriétaire de toutes les instances. D'autres utilisateurs peuvent créer des instances de classes mais c'est le *class administrator* qui régule leurs autorisations. C'est une vision plus centralisée.

Dans le modèle présenté, le choix d'une gestion centralisée ou décentralisée est proposé au créateur et premier propriétaire d'une classe lors de la création de la classe

Dans le deuxième modèle, l'accès aux vues doit se faire par deux types d'utilisateurs: les officiers chargé de la sécurité de base de données et les utilisateurs. Les officiers de la sécurité devront avoir un accès sans restrictions aux vues pour définir et maintenir la base de données. Les utilisateurs devront avoir des droits d'accès discrétionnaires aux vues pour respecter le principe de moindre privilège.

Agrégation et inférence

Agrégation

L'agrégation a lieu quand un utilisateur collecte des données de niveau de sécurité bas pour déduire la valeur de données hautement classifiées.

Appliqué au concept de view, il faut qu'une *view aggregate* d'autres view domine strictement leurs niveaux de sécurité. Il doit donc exister au moins une view dont le niveau est égal à celui de la vue *aggregate*.

Inférence

L'inférence arrive quand un utilisateur peut déduire de l'information en réunissant une collection d'*inquiries.* Les views représentent des relations entre certaines données, relations qui peuvent être exploitées. Cependant elles imposent des restrictions sur la visibilité des données et contraignent l'utilisateur à utiliser une petite partie de la base de données.

D'autres types d'inférence ne découlant pas de ces modèles sont basés sur la logique d'équation. Dans les bases de données orientées objet, d'autres modèles d'autorisation basés sur les méthodes régulent l'utilisation des objets : un objet peut être accédé seulement via ses méthodes. Une autorisation A pour un utilisateur u peut être représentée par un ensemble de droits m(c1,...,cn). L'utilisateur u peut employer ces méthodes directement sur des tuples d'objets disons (o1,...,on) si bien que oi appartient à une classe ci.

Cependant même si les méthodes m ne sont pas autorisées à u, u peut trouver un moyen d'invoquer m indirectement. On peut décrire ce phénomène en faisant une approche par une logique d'équations.

Prenons l'exemple de Employee, Host et Room, classes représentant respectivement les employés, les serveurs et les salles. On suppose alors qu'il existe une méthode computer qui retourne le serveur qu'un employé utilise et une méthode location qui retourne la salle ou le serveur se trouve. Puis une autre méthode office retournerait la salle occupée par un employé notée office(x) = location(computer(x)). On suppose également que la typologie du réseau est top secrète.

On suppose maintenant que u a réussi à obtenir que computer(John) = mars et office(John) = A626. Comme u réussi à connaître l'implémentation de office, il peut établir que location(computer(John)) = A626 et déduire que location(mars) = A626. Dans un schéma S de base de données, (c1,…..,cn) étant les classes, une composition de méthodes q(x1,...,xn) est dite sécurisée à (c1,...,cn) contre les attaques d'inférence par u si u ne peut déduire le résultat de q(o1,...,on) pour des objets oi de class ci dans n'importe quelle instance de base de donnée I de S en utilisant seulement des méthodes autorisées. Dans notre cas, si computer et location sont autorisées, location est insecure. Mais si computer n'est plus autorisée, et office et location le sont, u ne peut déduire que computer(John)=mars car il pourrait y avoir un autre serveur dans la salle A626 tel que computer (John) = Neptune.

Une solution serait d'approximer l'inférence de l'utilisateur et ce en établissant des règles de réécriture qui remplacent dans les méthodes les objets par leurs classes. Cette approximation de l'inférence par classe est un problème décidable.

CONCLUSION

Dans cette synthèse deux modèles d'autorisation dans les bases de données sont proposés. Le premier propose d'établir des règles dépendant du contenu des objets, il couvre les versions et objets composites et présente de nouveaux types d'autorisations (Read-all par exemple). Il permet de couvrir beaucoup d'autorisations grâce à ses autorisations implicites et permet 2 types d'administration des autorisations. Le deuxième modèle considère que les objets peuvent avoir plusieurs niveaux de sécurité. Il est une amélioration du système de vues dans une base de données grâce aux notions de contraintes de sécurité qu'on peut y ajouter. Il supporte également les classifications *content-dependent* et les vues peuvent également avoir plusieurs niveaux de sécurité pour des objets *single-level*. D'autres modèles tels que les modèles basés sur les méthodes des objets existent.

RÉFÉRENCES

[1] A multi-level view model for secure object-oriented databases par Ahmad Baraani-Dastjerdi, Josef Pieprzyk , Reihaneh Safavi-Naini

[2] An approach to authorization modeling in object-oriented database systems par E. Bertino et H. Weigand
[3] An equational logic based approach to the security problem against inference attacks on object-oriented database , par Yasunori Ishihara, Toshiyuki Morita, Hiroyuki Seki , Minoru Ito
[4] http://www.granddictionnaire.com/btml/fra/r_motclef/index800_1.asp

Cryptographie dans les bases de données

Thomas Courtois

INTRODUCTION

De nos jours les bases de données sont un des éléments sensibles des infrastructures informatiques. En effet elles regroupent souvent une quantité énorme d'information en un seul lieu, cela attirant les personnes mal intentionnées à la recherche de données sensibles. Pour les protéger il existe de nombreuses méthodes, le plus souvent externes à la base de données, bloquant l'accès aux utilisateurs non autorisés.

Le problème est que une fois ces défenses franchies, l'attaquant peut avoir accès à toutes les informations qu'il désire de manière très simple. De plus, il arrive régulièrement que les entreprises délocalisent leur base de données chez des hébergeurs externes et perdent ainsi le contrôle de leurs données. Il faut alors s'assurer que les personnes gérant ces hébergements ne puissent pas avoir accès au contenu des bases de données.

Une solution à ce problème est d'utilisé la cryptographie afin de protéger les données contenues dans la base. De cette façon les données restent secrètes même pour la personne contrôlant la base de données.

La cryptographie n'est pas une science nouvelle, mais cette dernière est de plus en plus utilisée dans la protection des données. Elle était jusqu'ici peu usitée dans les bases de données car son utilisation demandait beaucoup de ressources pour crypter et décrypter les informations à chaque requête. Mais avec l'évolution du matériel et des techniques de cryptage on trouve aujourd'hui des systèmes permettant une meilleure protection de ses données.

Dans une première partie nous allons voir quelles sont les divers solutions possible pour la crypter une base de données. Ensuite nous nous intéresserons une solution implémentée ayant pour but de protéger la confidentialité des données dans une base.

Problématique : Comment garantir la protection des données vis-à-vis d'un attaquant ou d'un administrateur trop curieux ?

On notera que sur les principes de base en sécurité, la cryptographie se focalisé sur la confidentialité des données et que leur intégrité et disponibilité passent en second plan.

CRYPTOGRAPHIE ET BASE DE DONNEES

Une base de données peut être divisée en plusieurs parties, les colonnes, les champs, les tables etc… Elles sont de plus généralement accessibles par plusieurs utilisateurs différents n'ayant pas forcément accès aux mêmes données. Les requêtes faites à une base peuvent être très varié selon l'utilisation qui sera fait des données par la suite.

On se rend alors compte que beaucoup de paramètres vont rentrer en jeu pour le cryptage d'une base de données.

Quelle partie crypter ?

La première réponse serait d'utiliser le même cryptage sur l'ensemble de la base, en la considérant comme une entité unique. Le problème est qu'il n'y a alors besoin de trouver qu'une seule clé pour accéder à toutes les données, il suffit alors d'avoir les droits pour accéder à un champ pour potentiellement avoir accès à tous les champs. Cette solution pourrait alors être envisageable dans les rare cas où la totalité de la base n'est accessible que par un utilisateur unique, supposé de confiance, ayant les droits d'accéder à l'ensemble des informations.

L'autre idée serait de crypter les colonnes séparément en leur attribuant des clés uniques. Cela permet de renforcer la sécurité, car si une colonne est compromise le reste de la table reste protégée. Mais ce gain de sécurité se fait au détriment des fonctionnalités, en effet il devient impossible d'effectuer une requête directement sur des informations cryptées différemment. Il faudra alors décrypter les informations d'une colonne puis de la seconde puis effectuer la requêtes et ensuite retourné le résultat. Et même pour le retour du résultat, il est généralement nécessaire que ce dernier soit crypté pour conserver le maximum de protection. Il faudra donc choisir quel cryptage utiliser pour le retour. Cela entraîne une grande consommation de ressource pour les requêtes complexes.

La situation est encore plus compliquée dans le cas où on choisirait de crypter chaque champ de la base avec une clé particulière.

Il faut également penser que la base de données doit stocker quelque part un index contenant un tableau permettant de relier les diverses clés de cryptage à diverses colonnes. Si cette table est stockée au sein de la base de données et que cette dernière est compromise l'ensemble du cryptage sera également compromis. La solution serait alors d'externaliser cette table mais cela demande d'augmenter l'infrastructure.

Gérer les utilisateurs et les clés de cryptage.

Dans la majorité des cas, lorsque l'on chiffre des informations il est nécessaire d'utiliser une clé, voire plusieurs dans certains cas, permettant d'avoir une protection efficace.

La solution la plus flexible est lorsqu'une donnée ne doit être accessible uniquement pour son propriétaire, est d'utiliser une clé qui sera liée à cet utilisateur. De manière général le système retenue est de créer une clé à partir du mot de passe de cet utilisateur ou alors pour éviter d'utiliser son mot de passe directement une variation de ce dernier. Il est alors assez simple de gérer les requêtes en rajoutant une partie précisant les champs à décrypter ou crypter et en fournissant la clé directement grâces aux informations utilisateurs que la base de données possède.

Les limites de cette technique sont principalement l'usurpation d'identité en se connectant à une base en se faisant passer pour un autre, et qu'elle n'est pas applicable pour des données qui doivent être accessible par plusieurs utilisateurs. En effet il est nécessaire d'avoir des clés de cryptages différentes selon les utilisateurs mais si deux utilisateurs peuvent avoir accès à la même information, cela veut dire que la même information devra être crypter deux fois, ou bien que la clé de cryptage devra être commune aux deux utilisateurs, ou encore que les utilisateurs aient accès à plusieurs clés de cryptage selon les données auxquelles ils ont besoin d'accéder.

Dans tous les cas cela complexifie grandement le système qui sera amené à gérer plus ou moins dynamiquement les clés de cryptage en fonction des droits d'utilisateurs.

Protéger les requêtes

Le dernier problème est lié à la confidentialité des requêtes. Dans certains cas il est nécessaire de cacher le contenu des requêtes au possesseur de la base de données afin qu'il ne puisse pas savoir sur quel partie travaille son utilisateur.

Nous allons maintenant étudier certaines solutions proposées de manière théorique.

PROPOSITION DE SOLUTIONS THEORIQUE

Cacher les requêtes à la base de données (PIR = Private Information Retrivial)

La première proposition est de cacher les informations sensibles dans une base de données en empêchant un attaquant de connaître quelles sont les requêtes effectuées par les utilisateurs.

Pour cela une solution est que le serveur envoie la base de données entière à la personne effectuant la requête. Puis la requête est ensuite traitée en local chez l'utilisateur en utilisant uniquement les informations nécessaires à la réalisation de cette requête.

Cette méthode ne répond pas à notre problématique car bien que protégeant la confidentialité des requêtes, le fait d'envoyer la base de données entière à l'utilisateur, nécessitant un utilisateur de confiance, rien n'empêche ce dernier d'être curieux et de chercher d'autre informations ne le concernant pas dans la base de données.

En plus de cela d'un point de vue efficacité cette méthode est très lourde car demande l'envoie d'une grande quantité de données puis d'un traitement en local.

Malgré tout cette solution peut être viable si elle est combinée à d'autre méthode de protection, ou bien si elle est utilisée dans des cas vraiment très spécifiques comme l'utilisation de petites bases de données.

Ré indexation des données dans la base

Une autre solution, dans le but de principalement cacher les requêtes effectuées par l'utilisateur, consiste à conserver les données de manière cryptée dans la base de données et d'utiliser un service de cartographie qui va utiliser une indexation particulière de façon à transformer la requête de l'utilisateur pour qu'elle soit comprise par la base de données contenant les données cryptées. Puis la base de données va renvoyer les données cryptées qui deviendront lisibles pour l'utilisateur après être repassé dans le système d'indexation.

L'avantage de cette technique est qu'elle permet de faire appel à des prestataires externes pour stocker les données sensibles de manière cryptée, sans avoir à investir en infrastructure, et de conserver la main sur le système de décodage en conservant la clé du coté client et non du coté serveur comme c'est habituellement le cas.

L'inconvénient de cette solution est qu'il est difficile de créer une bonne indexation pour toute une base de données, surtout si cette dernière est complexe. Le problème lié à cela est qu'il faut que cette indexation rende la cartographie suffisamment proche des données cryptées de façon à ne pas avoir à trop transformer les requêtes. Mais il ne faut pas non plus qu'une indexation trop proche permette de deviner les différences entre la requête normale et la requête modifiée et ainsi passer outre le système d'indexation en interceptant et décryptant les informations avant qu'elles arrivent au client.

Le cryptage brut des données

Il existe deux types principaux de cryptage direct des données, le cryptage symétrique et le cryptage asymétrique.

Le cryptage symétrique.

Le principe du cryptage symétrique est qu'il utilise une clé unique, privée, aussi bien pour le cryptage des données que pour leur décryptage. Cette clé est censée être unique et lié à la base de données. Le risque est alors que la clé soit divulguée par une personne mal intentionnée ou tout simplement craquée. La base de données devient alors totalement lisible. La difficulté est alors de maximiser la sécurité lors de la transmission de la clé afin qu'elle ait le moins de chance possible de tomber entre de mauvaises mains. Le principal avantage de cette méthode est qu'elle permet de donner un minimum de sécurité sans perdre trop en performances.

Les principaux algorithmes de cryptage symétrique sont le DES, le IDEA ou encore le AES.

Le cryptage asymétrique

Contrairement au système précédent utilisant une seul clé, celui-ci va utiliser deux clés distinctes. Une clé publique pour le cryptage des données et une clé privée pour le décryptage. Le principe de la clé privée est qu'elle soit impossible à deviner malgré la connaissance de la clé publique. La clé public est alors diffusée lors de l'envoie des données chiffrés et l'utilisateur possédant la clé privée, qu'il doit garder pour lui, va pouvoir l'utilisé avec la clé public avant d'inverser la fonction de chiffrement et obtenir les données en clair.

L'avantage de cette technique est qu'elle permet une bonne protection des données mais elle nécessite en contrepartie un long temps de calcul pour effectuer les opérations.

Les principaux algorithmes de cryptage asymétriques sont les algorithmes de type DSA/RSA.

Inconvénients

Cryptez directement les informations peut donc entrainer une perte de fonctionnalités car la base de données n'a plus l'accès total à ses données et cela risque de limiter les possibilités de requête.

De plus lorsqu'une requête doit gérer les données de plusieurs champs cryptés différemment, il est alors nécessaire de décrypter les informations avant même de pouvoir essayer d'effectuer la requête.

Le problème de ce type de protection est qu'elle est très gourmande en ressources, une opération de chiffrement ou déchiffrement est généralement assez longue à réaliser en comparaison d'autres opérations plus classiques, et avec ce système il est nécessaire d'effectuer une série de différents cryptages et décryptages pour obtenir le résultat d'une simple requête. La sécurité est probablement optimale mais la performance de la base est fortement impactée.

La solution qui a été proposée à ce problème est de gagner en performance en effectuant les requêtes directement avec les données cryptées et ainsi éviter de décrypter les informations pour effectuer la requête, le décryptage ne se fera uniquement sur le résultat. L'idée est alors de crypter chaque bit dans la base de données et ainsi effectuer les requêtes directement sur les bits cryptés.

Le problème est que pour que ce type de système fonctionne, il est nécessaire de réaliser une clé de cryptage fonctionnant de la même façon avec toute les portes logiques de base traitant les bit AND, NOR , NAND etc…

Si cette condition n'est pas respectée le système ne pourra pas effectuer toutes les requêtes.

Le hachage des données

Le hachage des données est une méthode assez répandue dans les bases de données afin de protéger les informations sensibles. Les algorithmes de ce type les plus souvent utilisés sont le MD5 ou le SHA-1. Le principe va être d'utilisation une fonction mathématique qui, pour une donnée en entrée, fournira un résultat unique en sortie, mais si un seul élément de la donnée d'entrée est modifié, le résultat en sortie sera très différent. L'avantage de cette technique est qu'elle permet de transformer par exemple un mot de passe de 6 à 8 caractères en une version cryptée d'une centaine de caractères, évitant les attaques par force brute.

Cette méthode est très souvent utilisée pour la protection des mots de passe dans les bases de données, en les cachant de manière assez simple.

Il suffit alors de comparer le mot de passe fournit pour l'authentification, en le faisant passer par une fonction de hachage, avec la version enregistrée du mot de passe dans la base, si les deux sont identiques, cela veut normalement dire que les mots de passe sont identiques.

Un des problèmes du hachage est le risque de collision. En effet les fonctions de hachages ont généralement un cardinale de l'ensemble de sortie, ou le nombre de résultats possibles en sortie de la fonction, borné. Dans le cas de MD5 par exemple le cardinal est de 16^32. Mais le cardinal de l'ensemble d'entrée, ou le nombre d'entrées potentielles, est lui infinie.

Il peut donc arriver que deux entrées totalement différentes fournissent la même valeur hachée.

Le défaut de cette méthode est qu'elle ne peut être généralisée partout sans être complexifiée, en développant ses propres fonctions de hachage par exemple. En effet comme le résultat d'une fonction de hachage est toujours le même si l'entrée est la même, il est possible de voir les répétitions dans une base de données. De plus même si une fonction de hachage n'est pas réversible, les pirates ont pu venir à bout de certaines fonctions de hachage en créant des tableaux géants de correspondance entre un mot et sa version hachée.

Malgré cela le hachage de données reste une des solutions les plus efficaces pour vérifier l'intégrité de données ou de fichiers et établir des signatures ou certificats électroniques.

Utilisation d'un catalogue de sécurité

Une des solutions proposée est de mettre en place, en parallèle de la base de données, un catalogue de sécurité.

Le principe de ce dernier est d'être impossible à modifier de manière manuelle, c'est-à-dire sans passer par des commandes directement dans la base et d'avoir un contrôle d'accès très strict pour l'atteindre.

Son rôle sera ensuite de garder une trace de tous les objets rajoutés dans la base en listant le nom de l'objet, son propriétaire, la façon de le modifier ou y avoir accès, la liste des personnes ayant les droit d'y avoir accès ainsi qu'idéalement une mémorisation de la dernière personne à l'avoir utilisé.

Si plusieurs personnes peuvent avoir accès aux mêmes données, le système enregistrera deux clés de cryptage distinctes liées à l'utilisateur, en utilisant une variation du mot de passe par exemple.

Ainsi ce fichier séparé sera la mémoire et le contrôleur pour l'accès aux données, cela permet d'éviter que si Bob essaye de se connecter avec la clé de cryptage d'Alice, le système lui refuse l'accès car il ne possède pas la bonne clé.

Dans la prochaine partie nous allons étudier un des principaux cas d'attaque que la cryptographie dans les bases de données essaye de contrecarrer, les attaques par inférence.

PROTECTION CONTRE L'INFERENCE

Dans cette partie les auteurs ont étudié quelle est la meilleure façon de protéger la base de données, hachage ou cryptage direct, contre deux types d'attaques par inférence.

Pour cette étude, la protection de la base de données s'organise en deux étapes. Le contenu de la base de données est chiffré, en utilisant dans un cas le cryptage direct et dans un autre cas une fonction de hachage, puis lors de l'envoie de la requête cette dernière est traduite de façon à cacher toutes les parties excepté les opérateurs, SELECT, FROM, AND etc. … De cette façon les noms de colonnes eux même sont cryptés et non identifiables. Ensuite l'application coté client va déchiffrer les informations renvoyées par la base et effectuer la requête sur les informations en clair.

Une des difficultés est que l'application côté client nécessite un certain niveau de complexité pour pouvoir effectuer ces opérations.

Le principal problème de ce système est sa vulnérabilité aux possibles attaques par inférence.

Qu'est-ce qu'une attaque par inférence ?

Le principe de ce type d'attaque est de se baser sur des informations connues pour essayer de « deviner » d'autres informations contenues dans la base de données. De cette façon le pirate peut arriver à reconstruire une base dans sa quasi-totalité en ayant qu'une partie des informations au départ.

L'équipe a testé différents scénarios pour vérifier l'efficacité de leur système, ils ont donc pris en compte deux types d'attaque.

La première attaque a pour but d'essayer de trouver certaines informations stockées dans la base de données en partant du principe que l'attaquant possède un accès à la base de données chiffrée ainsi qu'au système d'indexation.

Son objectif est alors d'essayer de découvrir les informations sensibles en décryptant la base de données.

La seconde attaque est différente car on considère que l'attaquant a accès à la base de données cryptée ainsi qu'à la base de données en clair et son but est cette fois ci de casser le cryptage du système d'indexation.

De cette façon l'attaquant sera par la suite capable de décrypter n'importe quelle information rajoutée dans la base de données ainsi que celles contenues dans des bases de données utilisant la même fonction d'indexation.

Lors de leur étude théorique et expérimentale les chercheurs ont tenté de déterminer quel était le meilleur moyen pour le cryptage d'une base de données. Ils ont donc pour cela étudié quatre scénarios distincts.

- Premier scénario : L'attaquant connait l'indexation et la base de données cryptée utilisant un cryptage direct

Ce scénario n'a pas été testé en conditions réelles car il représente une version simplifiée du second scénario.

En effet, l'équipe a prouvé lors d'une étude théorique que le comportement d'une base de données pouvait être déterminé de manière précise en analysant la cardinalité de ses attributs. Cela ne représentait donc pas d'intérêt du point de vue algorithmique.

- Second scénario : L'attaquant connait l'indexation et la base de données cryptée utilisant le hachage

- Troisième scénario : L'attaquant connait les bases de données non cryptée et cryptée utilisant le hachage

- Quatrième scénario : L'attaquant connait les bases de données non cryptée et cryptée utilisant un cryptage direct.

Pour ces trois derniers scénarios plusieurs séries de tests ont été effectuées dans différentes conditions. En faisant varier le nombre d'éléments dans la base, le nombre de colonnes ou encore des fonctions de cryptages différentes.

Résultats de cette étude

Les résultats suivants se basent sur plusieurs critères déterminés comme importants par l'équipe de chercheurs. Parmi ces critères on retrouve le facteur de collisions qui correspond dans une fonction de hachage au nombre de fois que deux mots différents peuvent avoir la même valeur hachée.

On retrouve le classique critère de performance de la table, qui est représenté par le temps moyen pour effectuer une requête.

Le risque de révélation, ou exposition, qui représente le degré de risque, pour la base de données, de subir une attaque par inférence.

- Pour le scénario 2 :

L'équipe a déterminé que malgré l'augmentation du temps de requête, liée à l'utilisation d'un index haché, les bénéfices en termes de protection contre les attaques par inférence étaient largement rentables. Cette perte de performance étant liée à la fonction de hachage utilisée, il est recommandé d'effectuer des tests afin de déterminer à partir de quelles valeurs le risque de révélation devient trop grand et nécessite de changer de fonction de hachage pour une fonction ayant un facteur de collision plus faible.

- Pour le scénario 3 :

Dans un premier temps les auteurs montre que le nombre d'attributs utilisés pour l'index ont un très grand impact sur le risque de révélation.

Il est également démontré que la taille de la base de données à une influence directe sur le risque de révélation. En effet la taille de la base permet à l'attaquant de plus facilement repérer des profils caractéristiques entre certaines informations notamment lorsque ces dernières sont redondantes. Mais en même temps la taille de la base de données rajoute un grand nombre de valeurs uniques très difficiles à identifier, et c'est cette caractéristique qui va prévaloir sur l'autre et rendre le risque de révélation plutôt faible.

- Pour le scénario 4 :

De nouveau le nombre d'attributs utilisés pour l'index est un des facteurs ayant le plus grand impact sur le risque de révélation, il faut donc les limiter au maximum dans le but d'assurer une bonne protection. De plus bien que le hachage permette de réduire de manière importante ce facteur, les valeurs d'exposition restent tout de même élevées malgré l'utilisation de fonction de hachage ayant un taux de collision faible.

Les chercheurs ont mis en évidence le fait que le risque de révélation diminuait plus le nombre d'éléments étaient important dans la base.

Ces scenarios ont montré que l'utilisation d'un simple cryptage direct est déjà une bonne protection dans de nombreux cas. La limite de ce type de cryptage est que lorsque le nombre d'index devient trop élevé la méthode perd beaucoup en efficacité.

La solution est d'alors de remplacer le cryptage en utilisant une fonction de hachage pour l'ensemble des index.

Ce qui ressort de manière globale de ces résultats est donc qu'il faut limiter le nombre d'attributs dans l'index et que la taille d'une base de données permet, plus elle est grande, de gagner en sécurité, dans un premier temps car plus la base de données est importante plus le temps de calcul nécessaire pour cracker son cryptage sera important mais également car les empreintes caractéristiques permettant habituellement de deviner les informations deviennent de moins en moins évidentes.

Bien sur ces résultats vont de pair avec la deuxième recommandation qui est de bien choisir sa fonction de cryptage ou de hachage selon la taille et le niveau de protection que l'on désire pour les informations contenues dans la base.

Au final l'équipe propose la mise en place, directement lors de l'élaboration des bases de données, d'un système de modèle permettant de calculer les performances de la base de données en fonction du nombre d'index utilisés et du type de cryptage.

Une telle simulation en amont de l'utilisation d'une nouvelle base de données donne la possibilité au développeur d'avoir une base ayant le meilleur compromis possible entre une sécurité optimale et une performance suffisamment grande pour ne pas gêner l'utilisateur.

Il n'y a donc pas de meilleure façon pour protéger sa base de données mais il faut trouver le meilleur compromis entre protection et performances. Pour cela l'étude propose de se baser sur différents facteurs, vus précédemment, aussi bien au niveau de l'utilisation de la base que des différentes fonctions de cryptages utilisées.

IMPLEMENTATION D'UNE SOLUTION : CRYPTDB

Le concept

Le principe de CryptDB repose sur deux éléments principaux. Le premier est un DBMS (DataBase Management System) non modifié qui contient des données qui sont cryptées ainsi que la table regroupant les clés cryptées. Ce DBMS aura également accès à une série de fonctions définies par l'utilisateur. L'ensemble est situé sur le serveur de DBMS. Le second élément est un serveur proxy pour la base de données, qui sera la passerelle entre le DBMS et l'application de l'utilisateur.

Dans ce système toutes les requêtes vont passer par le serveur de proxy, ce serveur va alors réécrire les requêtes pour qu'elles puissent s'effectuer directement sur les données cryptées. Le but de cette opération est d'empêcher la lecture de données cryptées sur le

serveur DBMS car ce dernier ne contient pas les clés permettant de décrypter les informations qu'elle contient. Le risque qu'un administrateur de base de donnée trop curieux ait accès à des informations privées est donc évité.

Dans le cas où le serveur de proxy et l'application client seraient également compromis, il serait impossible de conserver la confidentialité des données, car la clé de cryptage serait accessible à l'attaquant. Pour limiter ce problème, l'équipe a mis en place un système de clé unique liée au mot de passe de l'utilisateur. Le système va donc utiliser cette clé pour crypter les données appartenant à cet utilisateur

Grâce à cette méthode, les données pouvant être récupérées seront uniquement celles des utilisateurs actuellement connectés car leur clé est stockée sur le serveur proxy au moment de leur authentification.

On permet ainsi de garder une certaine confidentialité des données même dans le cas où la totalité du système est compromis.

Le but de CryptDB est d'assurer la confidentialité des données plus que leur intégrité ou leur disponibilité. En effet CryptDB ne pourra pas empêcher la modification ou même la suppression d'éléments dans la base de données mais elle permet d'assurer que les informations sensibles ne seront pas révélées à un attaquant ou un administrateur trop curieux.

Nous allons maintenant étudier comment fonctionne le mécanisme principal de défense utilisé par CryptDB. Les requêtes sur les données chiffrées.

Requêtes sur les données chiffrées

Dans cette partie on suppose que le serveur de proxy et l'application client n'ont pas été compromise, mais que le DBMS serveur peu l'avoir été.

On peut diviser la gestion des requêtes en quatre étapes.

1) Réécriture de la requête

Une fois que l'application du client envoie sa requête à la base de données, cette requête arrive au serveur proxy qui va se charger de réécrire cette dernière de façon à la rendre exécutable dans la base de données cryptée.

Pour cela le proxy va analyser la requête pour pouvoir définir quel sera le meilleur type de cryptage à effectuer sur les différents éléments de la requête. En effet certaines requêtes demandent des niveaux d'accès plus ou moins important, selon si il s'agit d'une simple récupération ou d'une opération complexe entre plusieurs champs.

Pour répondre au mieux à ce problème l'équipe de CryptDB propose de 6 méthodes de cryptage différentes afin de s'adapter à la situation.

- Méthode « Random »

Cette méthode est celle offrant la plus grande sécurité dans CryptDB, en effet ce système permet d'obtenir des cryptogrammes différents pour la même valeur grâce à un système probabiliste de cryptage.

Par contre ce système empêche la réalisation d'opérations de manière efficace entre les données.

Cette méthode est donc préférable dans le cas où l'on cherche uniquement à accéder de manière simple aux données, pour les autres cas elle n'est pas vraiment conseillé car trop lourde à gérer.

- Méthode Déterministe

Cette méthode reste assez proche de la précédente, la différence est qu'au lieu d'utiliser un algorithme probabiliste pour le cryptage, le système en utilise un déterministe. Cela veut donc dire que pour une deux valeurs identiques la valeur cryptée sera la même. De cette manière il devient possible de réaliser des vérifications d'égalité de manière très simple entre plusieurs champs.

On peut alors effectuer les opérations tel que GROUP BY, COUNT ou DISTINCT sans problème, tout en conservant une très bonne confidentialité des données

- Méthode OPE (Order-Preserving Encryption)

Le but de cette méthode de cryptage est de conserver la notion d'ordre entre les valeurs en claires et leur versions cryptées. Lors d'une requête nécessitant une notion d'ordre, le système va générer de manière aléatoire des valeurs cryptées tout en s'assurant que l'ordre n'est pas modifié.

De cette façon on rend possible les requête de type ORDER BY, MIN, MAX etc. mais la sécurité s'en retrouve diminuée car l'ordre étant conservé l'attaquant peut ainsi y avoir accès et s'en servir pour potentiellement briser le cryptage.

- Méthode JOIN et OPE-JOIN

La dernière étape avant l'accès à n'importe quelle donnée de la table est de rendre possible les jointures. Pour cela l'équipe à choisit de réaliser une nouvelle table de correspondance à part permettant de gérer les égalités entre les colonnes de la même façon que la méthode

déterministe. Et ils ont fait la même chose en adaptant la méthode OPE pour pouvoir effectuer des comparaisons d'ordre entre deux colonnes.

D'un point de vue sécurité ces méthodes ne sont en théorie pas plus dangereuses, mais étant donné qu'elle permette d'utiliser plusieurs colonnes, elles donnent accès à un plus grand nombre d'informations et donc diminuent le niveau de confidentialité du système.

- Méthode Homomorphique

Le but de cette méthode est de rendre possible les opérations telles que l'addition ou la multiplication de valeur. Le problème est que cette méthode est généralement très lente pour effectuer ces opérations sur des données cryptées.

- Méthode SEARCH

Enfin la dernière méthode utilisée est celle permettant la recherche de mots parmi les données cryptées. Ce type de recherche est utile pour les requête de type LIKE. Pour cela l'équipe effectue, pour chaque colonne ayant besoin d'un SEARCH, une séparation du mot en utilisant des mots clés, supprimant les répétitions de ces mots clés et permutant de manière aléatoire les mots clés restant. Cela en fait une des méthodes les plus sécurisées et permet d'effectuer des requêtes nécessitant LIKE.

2) Mise à jour de la requête grâce aux UDF

Une des forces de cryptDB est de permettre un ajustement du type de cryptage en fonction de la requête. De cette façon le système utilise toujours le niveau de cryptage maximum afin de réaliser la requête dans les meilleures conditions. Par exemple si la requête n'a pas besoin de vérifier une égalité ou un ordre particulier, le système utilisera automatiquement RND.

Pour cela l'équipe a eu comme idée de construire son système sous forme « d'oignons ». Pour cela ils ont décidé de crypter toute les données dans un ou plusieurs « oignons », chaque couche apportant un meilleur cryptage. Tous les oignons ne sont pas les mêmes selon les valeurs, par exemple pour une colonne contenant des INT la couche de cryptage permettant d'effectuer le SEARCH est inutile et ne sera donc pas enregistrée.

La couche la plus haute (RND) est celle offrant le moins de fonctionnalités mais le plus de protection, plus on avance et plus les couches permettent certaines fonctionnalités mais diminue la protection des données.

La question est alors de comment savoir sous quelle forme est actuellement enregistrée telle donnée dans la table ?

Pour cela le serveur proxy contient une table listant le niveau de cryptage actuel de chaque colonne. Et à chaque requête le système UDF va effectuer une requête préliminaire d'update sur la colonne visée afin de lui donner le mode de cryptage nécessaire à la réalisation de la requête de manière optimale.

Cette gestion des couches de protection de manière dynamique est effectuée du côté du serveur DBMS.

3) Le serveur proxy envoie la requête au serveur DBMS

La requête est maintenant réalisée sur le serveur DBMS qui va renvoyer une requête cryptée au serveur de proxy.

4) Le résultat est transmis à l'application client

Enfin le résultat de la requête cryptée est décrypté par le serveur proxy qui va la renvoyer à l'application cliente afin qu'il puisse l'utiliser.

Ce système de cryptage complexe permet donc de conserver le maximum de confidentialité dans le cas où le serveur DBMS est compromis. Le second problème se pose si le reste du système est compromis.

Gestion des « principaux »

Pour garder le maximum de confidentialité même lorsque que la totalité du système est compromis l'équipe a eu recours à un système de principaux afin de mieux protéger les données, ici un principal peut représenter un utilisateur unique ou un groupe d'utilisateur.

Il est donc possible avec CryptDB de créer des principaux et de définir à quelles partie de la base de données ces personnes ont accès, cette table sera alors stockée par le serveur proxy.

Le but est d'obtenir une clé unique de cryptage pour chaque principal. Pour cela le système va utiliser le mot de passe du principal afin de générer une clé de cryptage dérivée de ce mot de passe pour les différentes couches des protections de ses données.

Cette clé étant stockée sur le serveur proxy uniquement au moment de la connexion, si le système est totalement corrompu, l'attaquant ne pourra accéder uniquement aux données des principaux actuellement authentifiés sur le système car les clés des utilisateurs non identifiés ne seront pas disponibles car la clé est détruite à la déconnexion du principal.

Performances

Une fois l'implémentation de leur système effectuée l'équipe de chercheurs a effectué de nombreux tests pour vérifier si les performances de CryptDB restaient bonnes malgré l'ajout d'un grand nombre de sécurités.

On peut ainsi voir qu'au niveau du serveur en lui-même cryptDB n'est pas capable d'effectuer autant de requêtes qu'un serveur MySQL classique. C'est surtout lorsque le serveur utilise plus d'un seul processeur que la différence entre les deux systèmes devient importante.

En analysant le temps d'exécution des différentes requêtes on s'aperçoit que la différence du nombre de requêtes par seconde dépend du type de requête. Pour les requêtes Join ou Equality, MySQL et CryptDB sont très proche. En revanche pour le Sum ou l'Insert les écarts sont conséquents. Cela s'explique par le manque de flexibilité que possède CryptDB pour effectuer certaines opérations sur les tables.

Enfin si l'on considère le temps de réponse d'une requête HTTP utilisant des champs cryptés, les différences sont assez faibles et CryptDB obtient des résultats assez proches d'un serveur MySQL classique.

Il est donc intéressant de montrer ici que l'ajout des nombreuses fonctionnalités de protection de CryptDB, son serveur proxy, la modification des requêtes etc… ne modifie que de très peu les performances habituellement obtenues avec MySQL.

L'équipe obtient donc des performances plus que correctes pour un tel système tout en conservant une très bonne confidentialité des données. En effet les données obtenues via différentes attaques sont impossible à déchiffrer pour l'attaquant tant que son propriétaire n'est pas connecté à la base de données.

Un système prometteur

Bien que CryptDB permette de répondre à de nombreuses utilisations pour protéger une base de données, il manque encore certaines fonctionnalités pour qu'il puisse être utilisé à grande échelle.

Un des principaux soucis est que le système ne supporte pas tous les types d'opérations sur les bases de données. Il est alors possible d'effectuer ces opérations, généralement complexe, en les décomposant en opérations plus simples mais ces dernières font perdre en performance en plus de rajouter de la complexité au traitement des requêtes.

L'autre solution est de rajouter de nouvelles couches d' « oignons » afin de pouvoir traiter ces différents cas et les possibles nouveaux types de données.

CryptDB réussi malgré tout le pari d'augmenter fortement la confidentialité de données stockées dans une base sans y perdre trop en performance. Les principales améliorations seraient d'avoir un temps d'exécution des requêtes plus homogène avec MySQL quel que soit le type de requête exécutée. Ce point fait partie des axes d'amélioration sur lesquelles l'équipe travaille.

CONCLUSION

Lorsque l'on parle de sécurité nous nous retrouvons très souvent face à un dilemme entre sécurité et performance. Il est alors important de fixer ses objectifs pour parvenir à choisir la solution qui nous conviendrait le mieux. De cette façon il est alors possible de faire un compromis entre ces deux exigences.

Dans cette synthèse nous avons donc pu voir les différents moyens utilisés de nos jours en cryptographie dans les bases de données. Certaines solutions resteront théoriques et peu implémentées dans le monde réel car leurs performances sont trop faible et nuirait à l'utilisateur de manière trop importante. En revanche nous avons vu qu'il est parfois possible d'utiliser des systèmes donnant un bon compromis entre sécurité et performance.

C'est le cas de CryptDB qui fait le choix d'utiliser divers concepts différents de sécurité et cryptographie afin de répondre aux mieux aux exigences d'aujourd'hui.

En effet il est important de souligner qu'il existe rarement une solution parfaite et que le meilleur moyen de répondre à un problème complexe comme celui de la confidentialité dans les bases de données est d'utiliser plusieurs propositions théoriques et parvenir à les faire cohabiter de façon à les rendre le plus efficace possible.

Au cours de cette synthèse j'ai pu déterminer qu'actuellement un des moyens les plus courant et les plus efficace de protection était l'utilisation d'une partie tierce gérant l'exécution des requêtes en s'affranchissant des risques liés à l'usurpation d'identité.

D'un point de vue personnel je pense que cette solution est la meilleure car elle est applicable aussi bien dans le cas d'une base de données hébergée en interne ou en externe, en garantissant dans les deux cas une sécurité maximum. En revanche une telle solution soulève le problème de la sécurité de cette partie tierce qui devient par conséquent le maillon faible du système.

La complexité des bases de données, des requêtes effectuées et la gestion des utilisateurs sont des éléments qui rendent très difficile l'obtention d'une solution parfaite. Il faut donc être capable de faire le compromis entre la sécurité nécessaire, les fonctionnalités désirées et les performances.

La cryptographie dans les bases de données permet donc de conserver un maximum de confidentialité des données vis-à-vis d'un attaquant ou d'un administrateur malveillant. Malgré cela, il est toujours possible pour l'attaquant de modifier ou même détruire l'ensemble de la base de données, ce qui peut être tout aussi préjudiciable. Il est donc nécessaire de coupler la cryptographie à des technologies de contrôle d'authentification ou d'intégrité des données afin de garantir une sécurité optimale.

RÉFÉRENCES

- Alberto Ceselli, Ernesto Damiani, Sabrina De Capitani Di Vimercati, Sushil Jajodia, Stefano Paraboschi, Pierangela Samarati: Modeling and Assessing Inference Exposure in Encrypted Databases
- Ueli Maurer: The Role of Cryptography in Database Security
- Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan: CryptDB: Protecting Confidentiality with Encrypted Query Processing
- Jingmin He Min Wang: Cryptography and Relational Database Management Systems

Conclusion

Dans cette synthèse, nous avons vu différents aspects de la sécurité des bases de données. Nous avons vu comment établir une politique de contrôle d'accès efficace avec les rôles, puis nous avons présenté des moyens pour mettre en place ce type de contrôle. Ensuite, nous avons vu comment mettre en place des modèles pour établir des vues contrôlant l'accès aux objets et des règles s'appliquant à ces objets dans le cas des bases de données orientées objets. Enfin, nous avons étudié comment assurer la confidentialité des données en utilisant la cryptographie.

Ces domaines ne couvrent que certains aspects de la sécurité des bases de données, et ne se veulent pas exhaustifs. Ils permettent cependant de mieux appréhender certains concepts clés.

Pour aller plus loin dans la sécurité dans les bases de données il pourrait être intéressant d'étudier les mécanismes permettant de traiter par exemple la disponibilité des bases de données, via par exemple la mise en place de base de données miroirs ou encore la mise en place de sauvegardes automatiques.