Formalisation et implantation d'une politique de sécurité d'une base de données

J. Blond & C. Morisset

Laboratoire d'Informatique de Paris 6,
8 rue du Capitaine Scott 75015 Paris, France
`blond@bertin.fr`
`charles.morisset@lip6.fr`

Résumé

La conception d'un programme chargé du contrôle d'accès d'un système gérant de l'information doit s'effectuer selon des règles de programmation strictes, afin d'éviter la présence de failles dans de ce programme. L'Atelier Focal est un environnement permettant d'écrire, au sein d'un même cadre de travail, des spécifications formelles et du code fonctionnel respectant ces spécifications. Nous avons utilisé cet outil pour formaliser une politique de contrôle d'accès, celle de Bell & La Padula, et l'implanter dans MySQL, un système de gestion de base de données. Cette implantation, nécessitant la traduction des requêtes SQL en termes d'accès Bell & La Padula, se fait sous la forme d'un module externe à la base de données qui filtre les requêtes, et rejette ainsi celles qui violent la politique de sécurité.

Introduction

Les systèmes multi-utilisateurs doivent définir des règles d'accès aux données. Il doit donc exister dans de tels systèmes un programme chargé de mettre en œuvre cette politique de sécurité, et qui doit vérifier que cette dernière ne sera pas violée. Ce programme doit répondre à des exigences très strictes de correction, car chacune de ses failles pourrait être exploitée par des attaquants pour violer la politique de sécurité dans le système complet.

Les Critères Communs fournissent une méthodologie permettant d'atteindre un haut niveau de sûreté. Ces critères constituent un recueil de normes définies par des agences gouvernementales (Orange Book, ITSEC, TCSEC). Ils définissent à la fois un cadre de travail pour la conception et la réalisation de logiciels et une réference pour les utilisateurs de ces logiciels.

Les hauts niveaux de sécurité des Critères Communs (EAL 5 à 7) requièrent l'utilisation de méthodes formelles dans les étapes de spécification et de conception du logiciel. Or, l'interaction de la plupart des langages impératifs avec les méthodes formelles est complexe et se limite le plus souvent à de la vérification. C'est pour cette raison que nous avons utilisé l'Atelier Focal [8]. Dans celui-ci, on peut écrire une spécification formelle de logiciel à developper puis, par des étapes de conceptions successives, arriver à du code fonctionnel. Ainsi, avec cet outil, il est possible d'écrire des propriétés en logique du premier ordre, d'écrire des fonctions dans un langage proche de OCaml, et de faire des preuves avec Zenon qui sont extraites vers Coq. On peut alors obtenir progressivement une implantation efficace à partir d'une spécification formelle.

Dans un précédent article [3], nous avons formalisé dans Focal l'algèbre de sécurité de McLean [5]. Cette algèbre fournit une définition générale de la notion de contrôle d'accès. Le modèle de Bell & La Padula [1] peut être vu comme une instance de cette algèbre. Dans cet article, nous montrons comment obtenir une implantation conforme au modèle de Bell & La Padula dans un système de gestion de base de données, à partir de la formalisation de cette algèbre dans Focal, .

Tout d'abord, nous présentons l'algèbre de McLean et nous montrons comment l'instancier pour obtenir le modèle de Bell & La Padula (section 1). Pour utiliser ce modèle, il faut pouvoir spécifier les accès caractérisant chaque requête SQL. Nous formaliserons donc les requêtes SQL de base en terme d'accès (section 2). L'architecture générale de l'implantation se présente comme un module externe qui filtre les requêtes SQL fournies par l'utilisateur (section 3).

1. Algèbre de McLean

1.1. Présentation

L'algèbre de sécurité de McLean est définie dans [5].

Elle fournit un cadre général dans lequel de nombreux modèles de contrôle d'accès peuvent être implantés. Un système est composé d'entités actives (les sujets) et d'entités passives (les objets). Les sujets accèdent aux objets selon certains modes (par exemple lecture et écriture). Le contrôle d'accès consiste à n'autoriser qu'un certain ensemble d'accès. Pour cela, chaque sujet et objet est qualifié par un niveau de sécurité (par exemple `Secret` ou `Confidentiel`). Les accès autorisés sont définis en fonction des niveaux de sécurité des sujets et objets, ainsi que des autres accès des sujets.

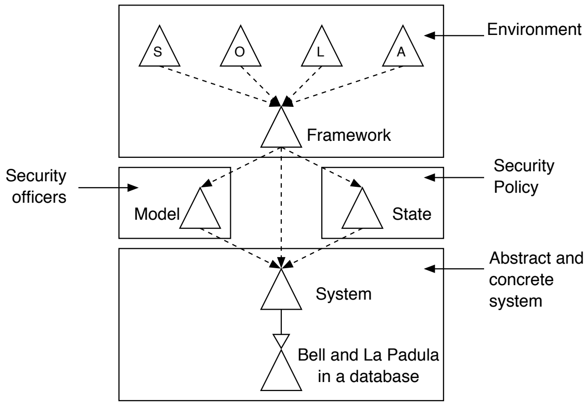

Figure 1: L'algèbre de sécurité

L'algèbre de sécurité est définie en trois étapes que nous présentons brièvement.

1.1.1. Frameworks

Un framework $\mathcal{F}$ est un quadruplet $(\mathcal{S}, \mathcal{O}, \mathcal{L}^S, \mathcal{A})$ où $\mathcal{S}$ est un ensemble fini de sujets (utilisateurs, programmes, ...), $\mathcal{O}$ est un ensemble fini d'objets (données, fichiers, ...), $\mathcal{L}^S = (\mathcal{L}, \preceq, \curlyvee, \curlywedge)$ est un treillis fini de niveaux de sécurité et $\mathcal{A}$ est un ensemble fini de modes d'accès (par la suite, $\mathcal{A}^\star$ sera utilisé pour indiquer l'ensemble des parties de $\mathcal{A}$). Chaque framework introduit un ensemble $\mathcal{S}^+$ inclus dans $\wp(\mathcal{S})$ contenant les ensembles de sujets qui peuvent demander à effectuer un accès conjoint. La notion d'accès conjoint n'est pas commune à tous les modèles de contrôle d'accès, mais elle peut être nécessaire dans certains domaines, comme par exemple le monde militaire, où l'on peut imaginer que pour lancer un missile, il faut que deux personnes appuient en même temps sur le même bouton. Néanmoins, comme nous le verrons par la suite, il est toujours possible de restreindre l'ensemble $\mathcal{S}^+$ à l'ensemble des singletons de $\mathcal{S}$, afin de retrouver la notion d'accès classiqe.

De plus, comme nous le verrons par la suite, un framework doit spécifier un ensemble $\mathcal{S}^{+\star}$, inclus dans $\wp(\mathcal{S}^+)$, d'ensembles d'ensembles de sujets (voir modèles).

1.1.2. Modèles

Etant donné un framework $\mathcal{F}$, un modèle est défini par deux fonctions c_s et c_o, respectivement de $\mathcal{S}$ et $\mathcal{O}$ vers $\mathcal{S}^{+\star}$.

c_s (resp. c_o) associe à chaque sujet (resp. objet) l'ensemble des différents groupes de sujets autorisés à demander un changement de son niveau de sécurité. Cet ensemble est donc un élement de $\mathcal{S}^{+\star}$.

En excluant certains élements de $\mathcal{S}^{+\star}$, il est possible d'exprimer des propriétés sur le système. En effet, si un groupe de sujet n'appartient pas à $\mathcal{S}^{+\star}$, il ne pourra demander la modification d'aucun niveau de sécurité. Par exemple, imaginons que le système contient un administrateur, identifié par le sujet *admin*. Pour imposer que toute demande de changement de niveau de sécurité doit se faire en accès conjoint avec l'administrateur, on peut définir $\mathcal{S}^{+\star}$ ainsi :

$$\mathcal{S}^{+\star} = \{X \subseteq \mathcal{S}^+ \mid \forall y \in \mathcal{S}^+ \; y \in X \Rightarrow admin \in y\}$$

1.1.3. Systèmes

Nous avons enfin la notion de système, où la politique de sécurité et la fonction de contrôle d'accès sont définies. Etant donné un framework $\mathcal{F}$ et un modèle $\mathcal{M}$, un système peut être vu comme une machine à état abstraite. L'ensemble Σ des états est défini comme le produit $\mathbb{F} \times \mathbb{A} \times \mathbb{A}$ où $\mathbb{F}$ est le produit $\mathbb{F}_s \times \mathbb{F}_o$ avec $\mathbb{F}_s = \{f_s : \mathcal{S} \to \mathcal{L}^S\}$ (niveau de sécurité associé à un sujet) et $\mathbb{F}_o = \{f_o : \mathcal{O} \to \mathcal{L}^S\}$ (niveau de sécurité associé à un objet), et $\mathbb{A} = \wp(\mathcal{S}^+ \times \mathcal{O} \times \mathcal{A}^\star)$ est utilisé pour décrire les droits discrétionnaires et les accès courants.

Un état $\sigma = ((f_s, f_o), d, m)$ définit les droits "mandatoires" (f_s et f_o, également appellés vecteur de classification) droits discrétionnaires (d) et accès courants (m). Pour pouvoir spécifier la politique de sécurité avec les accès conjoints, nous étendons f_s comme suit:

$$\begin{array}{ll} f_s^{\curlywedge} : \mathcal{S}^+ \to \mathcal{L}^S & f_s^{\curlyvee} : \mathcal{S}^+ \to \mathcal{L}^S \\ f_s^{\curlywedge}(S) = \curlywedge\{f_s(s) \mid s \in S\} & f_s^{\curlyvee}(S) = \curlyvee\{f_s(s) \mid s \in S\} \end{array}$$

A ce niveau, nous pouvons seulement définir les propriétés de sécurité sur les états. Etant donné que nous voulons implanter une politique de confidentialité, (l'information contenue dans les objets ne peut être lue que par les sujets autorisés), nous spécifions ici les trois propriétés classiques de cette politique :

- un état est dit ds-sûr si et seulement si: $\forall S \in \mathcal{S}^+$, $\forall o \in \mathcal{O}$, $(S, o, x) \in m \Rightarrow (S, o, x) \in d$
- un état est dit simple-sûr si et seulement si: $\forall (S, o, A) \in m$, $\mathsf{read} \in A \Rightarrow f_s^{\curlywedge}(S) \succeq f_o(o)$
- un état est dit $\star$-sûr si et seulement si:

$$\forall S_1, S_2 \in \mathcal{S}^+ \; \forall o_1, o_2 \in \mathcal{O} \; \forall e_1, e_2 \in \mathcal{A}^\star$$
$$\left(\begin{array}{lll} & (S_1, o_1, e_1) \in m & \wedge \;\; \mathsf{read} \in e_1 \\ \wedge & (S_2, o_2, e_2) \in m & \wedge \;\; \mathsf{write} \in e_2 \\ \wedge & S_1 \cap S_2 \neq \emptyset & \end{array} \right) \Rightarrow f_o(o_1) \preceq f_o(o_2)$$

  Cette propriété permet d'éviter la copie d'un objet à un niveau de sécurité plus bas par un sujet. (voir figure 2). Cette propriété est légèrement différente de celle définie dans [5]. En effet lorsqu'il n'y a pas d'accès conjoints, mais uniquement des accès effectués par des sujets seuls, nous voulons retrouver la définition classique de la propriété "MAC-$\star$" définie par Bell and La Padula dans [1]. Or ce n'est pas la cas définie par McLean.

Un état vérifiant ces trois propriétés est dit *sûr*.

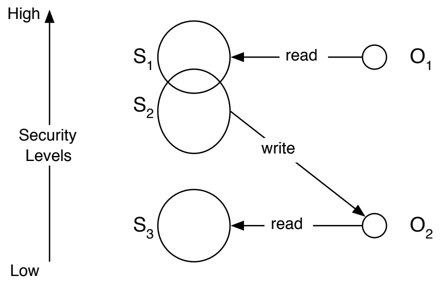

Figure 2: Violation de la sécurité-$\star$

1.1.4. Fonction de transition

Une fonction de transition est une fonction $\tau : \mathcal{S}^+ \times \mathcal{R} \times \Sigma \to \mathcal{D} \times \Sigma$ avec Σ l'ensemble des états, $\mathcal{R}$ l'ensemble des requêtes et $\mathcal{D}$ l'ensemble des réponses.

Une fonction τ est dite sûre si et seulement si:

$$\tau(S_1, r, \sigma_1) = (d, \sigma_2) \Rightarrow \left(\begin{array}{ll} & \forall s \in \mathcal{S} \quad f_s^1(s) \neq f_s^2(s) \Rightarrow S_1 \in c_s(s) \\ \wedge & \forall o \in \mathcal{O} \quad f_o^1(o) \neq f_o^2(o) \Rightarrow S_1 \in c_o(o) \end{array} \right)$$

En d'autres termes, si le niveau de sécurité d'une entité a changé, alors l'ensemble de sujets ayant initié ce changement était autorisé à le faire.

1.2. Bell & La Padula

Le modéle de Bell & La Padula [1] peut être vu comme une instance de l'algèbre de McLean.

1.2.1. Instanciation de l 'algèbre

Nous allons maintenant expliquer comment obtenir le modèle de Bell & La Padula à partir de l'algèbre de McLean. Nous définissons tout d'abord un framework où $\mathcal{S}^+$ ne contient que des singletons, ce qui signifie qu'il n'y a pas d'accès conjoints. Nous définissons également $\mathcal{L}^S$ comme le treillis produit $T_c \times T_k$ où $T_c = (\mathcal{C}\mathrm{l}, \leq, \sqcup_{cl}, \sqcap_{cl})$ est le treillis des classifications et $T_k = (\wp(\mathcal{K}), \subseteq, \cup, \cap)$ est l'ensemble des parties finies de l'ensemble $\mathcal{K}$ des besoins-d'en-connaître, et $\mathcal{A} = \{$read, write , execute , append ,control $\}$.

Le modèle de Bell & La Padula est celui où $c_s(s) = c_o(o) = \mathcal{S}^+$ pour tout sujet s et tout objet o [1]. Il est clair qu'avec un tel modèle, toute fonction de transition sera sûre. Pour définir le système, il faut définir les requêtes, les réponses et la fonction de transition. Cela a été fait en suivant [4] ainsi qu'en réutilisant la formalisation de ce système dans Coq [7] précédemment réalisée dans [2].

1.2.2. Fonction de transition de Bell & La Padula

Nous donnons ici une brève description de la fonction de transition définie par Bell & La Padula. La définition complète peut être trouvée dans [2].

Prenons comme exemple $\mathcal{S} = \{Jean,\ Anne\}$ et $\mathcal{O} = \{F12.tex,\ F56.ps\}$.

Une requête est un 5-uplet (s_1, t, s_2, o, p),

[1] Il existe une version appelée "Bell and La Padula with tranquility" où $c_s(s) = c_o(o) = \emptyset$ pour tout sujet s et tout objet o.

où s_1 est l'ensemble de sujets initiant la requête, t est le type de la requête (G, R, C ou D), s_2 est l'ensemble de sujets concerné par la requête, o est l'objet concerné par la requête et p est le paramètre de la requête.

s_1 est généralement vide, sauf dans les cas où il est réellement important de connaître l'ensemble de sujets ayant initié la requête. Par exemple, dans Bell & La Padula, un sujet peut donner ses droits sur un objet à un autre sujet. Dans ce cas, s_1 est le premier sujet et s_2 le second. Un autre exemple est lorsqu'un sujet veut changer le niveau de sécurité d'une entité. Dans ce cas, s_1 sera le sujet, et s_2 sera le sujet vide (ce que l'on note par σ_ϵ).

Les requêtes de type G concernent les demandes d'accès. Si *Jean* veut lire *F56.ps*, il créera la requête suivante : $(\sigma_\epsilon, G, \textit{Jean}\ , \textit{F56.ps}\ , \chi_{\mathcal{A}}(\mathsf{read}))$.

Les requêtes de type R concernent le relâchement d'accès. Donc quand *Jean* a fini de lire *F56.ps*, il soumet la requête suivante $(\sigma_\epsilon, R, \textit{Jean}\ , \textit{F56.ps}\ , \chi_{\mathcal{A}}(\mathsf{read}))$.

Les requêtes de type C concernent la création d'objets et l'affectation des niveaux de sécurité. Par exemple, si *Anne* veut créer le fichier *F32.tex*, elle envoie la requête suivante $(\sigma_\epsilon, C, \textit{Anne}, \textit{F32.tex}\ , \chi_\epsilon)$. Elle lui donne ensuite la classification *Secret* et l'ensemble de besoins-d'en-connaître *{OTAN, Marine}* avec $(\textit{Anne}, C, \sigma_\epsilon,\ \textit{F32.tex}\ , \chi_{\mathcal{F}}(\textit{F32.tex}\ , \textit{Secret}\ , \{\textit{OTAN, Marine}\ \}))$.

Les requêtes de type D concernent la destruction d'objets. Si *Anne* veut détruire *F56.ps* elle soumet la requête $(\sigma_\epsilon, D, \textit{Anne}\ , \textit{F56.ps}\ , \chi_\epsilon)$.

Chaque type de requête est exécuté par une fonction différente. La fonction t_G exécute les requêtes de type G, t_R celle de type R, t_C celle de type C et t_D celle de type D.

2. Sémantique de SQL

2.1. Demande et relâchement des accès

Lorsque l'on veut définir une sémantique pour SQL en terme d'accès dans le modèle de Bell & La Padula, il faut prendre en compte la notion de persistance des accès. En effet, dans le modèle de Bell & La Padula, chaque accès se décompose en deux phases : premièrement, on demande l'accès, deuxièmement on relâche l'accès. Tant qu'un accès n'est pas relâché, il reste dans l'ensemble des accès courants. Cette caractéristique permet de modéliser le fait que dans certains systèmes, comme par exemple un gestionnaire de fichiers, les accès sont persistants.

Dans une base de données SQL, une requête n'est pas persistante, du fait de la structure client/serveur. En effet, un client soumet une requête, le serveur analyse cette requête, ouvre les objets correspondants, lit ou écrit dans ces objets, ferme l'objet et renvoie la réponse au client.

Nous définissons donc deux relations $<>_g$ et $<>_r$: $< R >_g$ effectue la demande d'accès pour la requête R et $< R >_r$ effectue le relâchement d'accès pour la requête R.

Nous allons voir maintenant comment fonctionnent ces deux relations avec l'exemple de la requête SQL `INSERT`, qui permet de rajouter des entrées dans une table.

2.2. Une sémantique de SQL en termes d'accès

La sémantique complète de SQL en terme d'accès Bell & La Padula est en annexe B.

2.2.1. Syntaxe abstraite

La syntaxe SQL de la requête `INSERT` est la suivante :

`INSERT [OR conflict-algorithm] INTO` *table-name* `[(`*column-list*`)] VALUES(`*value-list*`)`

`| INSERT INTO` *table-name* `[(`*column-list*`)]` *select-statement*

Nous supposons que l'utilisateur soumettant cette requête est connu et que son identificateur est disponible dans l'environnement global. Etant donné que la granularité des objets est la table (voir 3.2.1), nous n'avons pas besoin de la liste des colonnes. De plus, si la requête `INSERT` est utilisée avec `VALUES`, nous n'avons pas non plus besoin des valeurs exactes pour définir l'arbre de syntaxe abstraite.

La syntaxe abstraite de la requête `INSERT` est donc:

$$INS(o, sel)$$

où o est l'objet correspondant à la table où l'insertion est effectuée, et sel est soit $\perp$ quand la requête est de la forme `INSERT INTO ...VALUES ...`, soit le terme de syntaxe abstraite de *select-statement* sinon.

2.2.2. Demande d'accès : $<>_g$

Etudions maintenant l'évaluation de la requête `INSERT` par $<>_g$. Si l'évaluation par $<>_g$ de *select-statement* est négative, alors l'évaluation du `INSERT` par $<>_g$ doit également être négative :

$$(INS_g^1)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}{< INS(o, sel), s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}$$

où s est le sujet soumettant la requête, c son niveau de sécurité et ρ l'état Bell & La Padula courant.

En revanche, si l'évaluation de *select-statement* par $<>_g$ est positive et renvoie donc un nouvel état ρ', alors le résultat est la réponse retournée par la fonction t_G avec une demande d'accès en ajout et l'état ρ', ce qui donne la règle d'inférence suivante :

$$(INS_g^2)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{yes}, \rho')}{< INS(o, sel), s, c, \rho >_g \rightarrow t_G((\sigma_\epsilon, G, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{a})), \rho')}$$

2.2.3. Relâchement d'accès : $<>_r$

De même pour le relâchement de la requête `INSERT`, si l'évaluation par $<>_r$ de *select-statement* est négatif, alors l'évaluation par $<>_r$ doit être négative.

$$(INS_r^1)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}{< INS(o, sel), s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}$$

En revanche, si l'évaluation par $<>_r$ de *select-statement* est positive, alors l'évaluation par $<>_r$ de la requête `INSERT` est le résultat renvoyé par la fonction t_R de Bell & La Padula demandant de relâcher l'accès en ajout de l'utilisateur sur la table concernée.

$$(INS_r^2)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{yes}, \rho')}{< INS(o, sel), s, c, \rho >_r \rightarrow t_R((\sigma_\epsilon, R, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{a})), \rho')}$$

Lors de l'évaluation d'une requête R, il est clair qu'il faut d'abord appeler $< R >_g$, puis $< R >_r$. En revanche, la question est plus complexe lors de l'évaluation de plusieurs requêtes en séquence. A première vue, si on veut évaluer une séquence de requêtes il suffit d'évaluer chaque requête une par une, indépendamment des autres. Mais comme nous allons le voir, cela peut créer des canaux cachés. Nous allons donc étudier trois différents modes d'évaluation.

2.3. Modes d'évaluation

Pour ne pas violer la propriété de sécurité-$\star$, une requête de la sorte :

`INSERT INTO` *table1* `SELECT * FROM` *table2*

doit être refusée si le niveau de sécurité de *table1* strictement inférieur à celui de *table2*. En effet, dans le cas contraire, de l'information de haut-niveau (*table2*) se retrouverait dans un objet de bas-niveau (*table1*).

Avec la définition de la relation $<>_g$, cette requête est effectivement refusée. En effet, lors de l'évaluation de cette requête, on effectue d'abord l'évaluation par $<>_g$ du `SELECT`, et sans relâcher cet accès, on évalue ensuite le `INSERT`, ce qui est logiquement refusé.

Mais imaginons maintenant que l'utilisateur soumette successivement les requêtes suivantes :

R_1 : `SELECT * FROM` *table2*;
R_2 : `INSERT INTO` *table1* `VALUES` val_1, ..., val_n

et que les val_i soient les valeurs que l'utilisateur lit sur l'écran et donc proviennent du `SELECT`. Il est clair que cette séquence de requêtes a le même effet que la requête imbriquée donnée ci-dessus et donc cette séquence devrait être refusée. Regardons à présent trois différents modes pour évaluer cette séquence.

2.3.1. Mode requête

Le premier mode est l'évaluation requête par requête :
$< R_1 >_g < R_1 >_r < R_2 >_g < R_2 >_r$

Le `SELECT` est évalué, et si il est correct, alors il est relâché. Donc lorsque l'on évalue le `INSERT`, on ne se souvient pas du `SELECT`, et donc le `INSERT` est accepté.

Cette séquence est donc acceptée dans ce mode d'évaluation, alors qu'elle ne devrait pas l'être. Nous sommes dans le cas typique d'un canal caché, en l'occurrence l'écran.

2.3.2. Mode strict

Le second mode est d'évaluation consiste à ne jamais relâcher les accès :
$< R_1 >_g < R_2 >_g$

Dans ce mode, la séquence est donc refusée. Cette option évite certes les canaux cachés, mais elle est excessivement restrictive. Par exemple, si un utilisateur exécute une requête `SELECT` sur un objet avec le plus haut niveau de sécurité, alors il ne pourra plus jamais effectuer de `INSERT` sur une table de niveau inférieur.

Donc ce mode doit être reservé aux bases de données extrêmement critiques.

2.3.3. Mode session

Etant donné que le mode requête crée des canaux cachés et que le mode strict est beaucoup trop restrictif, nous introduisons le mode session où chaque requête est relachée dans l'ordre inverse d'exécution :
$< R_1 >_g < R_2 >_g < R_2 >_r < R_1 >_r$

Suposons qu'un utilisateur exécute un SELECT puis un INSERT, le SELECT est évalué, et s'il est correct, il n'est pas relâché tout de suite. On évalue ensuite le INSERT et s'il est correct, on peut alors relâcher le INSERT puis le SELECT Ce mode d'évaluation effectue les appels aux relations $<>_g$ et $<>_r$ dans le même ordre que si les deux requêtes étaient imbriquées dans une unique reqête.

De manière générale, lorsque l'utilisateur se connecte, toutes ses requêtes sont évaluées à l'aide de $<>_g$ et les appels correspondants à $<>_r$ sont empilés. Lorsque l'utilisateur quitte la session, la pile des $<>_r$ est vidée.

Il faut toutefois que l'environnement d'exécution soit "nettoyé" après la fin de session de l'utilisateur. En effet, si ce dernier exécute un SELECT et se déconnecte, mais que le résultat est toujours affiché à l'écran, il pourrait alors se reconnecter et ainsi exécuter un INSERT avec les valeurs affichées. Il pourrait donc violer la politique de sécurité sans que cela soit détecté. Il faudrait donc avoir un mécanisme permettant d'éviter de telles situations, et qui pourrait par exemple effacer l'écran ainsi que les fichiers temporaires créés.

3. L'atelier Focal

3.1. Présentation

Le langage Focal permet de spécifier, prouver et implémenter. Il utilise les notions d'espèce, de collection, de type support et d'héritage multiple. Nous rappelons brièvement ces notions à l'aide de l'exemple bien connu des entiers.

```
species setoid inherits basic_object =
  sig equal in self -> self -> bool;
  sig element in self;
  let different(x,y) = #not_b(equal(x,y));
  property equal_reflexive = all x in self, !equal(x,x);
  property equal_symmetric = all x y in self, !equal(x,y) -> !equal(y,x);
  property equal_transitive = all x y z in self, !equal(x, y) -> !equal(y, z)
     -> !equal(x, z);
end
```

Le listing précédent montre que l'espèce `setoid` hérite de l'espèce `basic_object`, cette dernière ne faisant que déclarer le `rep` et définir les méthodes `print` et `parse` par défaut. Le `rep` représente le type support de l'espèce c'est-à-dire le type des éléments de l'espèce. Par la suite, ce type est représenté par le mot clef `self` dans le type des méthodes ou des arguments. Dans `setoid`, il n'est toujours que déclaré mais on peut y faire référence comme dans le type de la méthode `equal`. Nous remarquons ensuite deux méthodes déclarées mais non définies : `equal` et `element`, et une autre définie : `different` comme étant la négation de l'égalité. À cela s'ajoutent trois propriétés (non prouvées) sur la méthode `equal` : `reflexivity`, `symmetry` et `transitivity`. À présent, nous définissons l'ensemble `monoid` :

```
species monoid inherits setoid =
  sig times in self -> self -> self;
  sig one in self;
  let element = !times(!one,!one):
  property times_associative : all x y z in self,
    !equal(!times(x,!times(y,z)),!times(!times(x,y),z));
end
```

Les monoïdes héritent de la structure de `setoid` (donc de ses méthodes). Nous rajoutons les méthodes `times` et `one` (loi multiplicative et élément neutre) et nous définissons `element` comme étant le produit de `one` avec lui même. Nous pouvons décider d'implémenter les entiers, pour cela, nous définissons la collection entier :

```
collection entiers implements monoid =
  rep = int;
  let one = 1;
  let times = #int_mult;
  let equal = #int_eq;
  let print = #string_of_int;
  let parse = #int_of_string;
  proof of equal_reflexive = assumed;
  proof of equal_symmetric = assumed;
  proof of equal_transitive = assumed;
  proof of mult_associative = assumed;
end
```

Une collection correspond à une espèce complètement définie, c'est-à-dire que toute les méthodes sont définies et toutes les propriétés prouvées. La collection `entiers` implémente `monoid` ce qui signifie que les entiers auront la structure de monoïde. Nous définissons donc le `rep` comme étant les entiers machine `int`, l'élement `one` comme valant 1, et ainsi de suite. Les preuves de reflexivité, de symétrie et de transitivité pour l'égalité sont déclarées comme `assumed`. Ce mot clé permet d'admettre une propriété dans le cas où elle n'est pas prouvable en l'état ou pour poser un axiome (c'est le cas ici). De plus, une collection est encapsulée. Cela signifie que l'utilisateur de cette collection n'a accès ni à son type support, ni aux définitions des méthodes, ni aux preuves des propriétés.

3.2. Implémentation

3.2.1. Définition du Framework

Nous reprenons donc la formalisation du modèle de Bell & La Padula dans Focal comme décrit dans 1.2. Il faut alors définir les espèces correspondants aux sujets, aux objets et aux classifications, les besoins d'en connaître étant définis comme des chaînes de caractères.

Nous implantons l'espèce des sujets avec des entiers, et nous créeons une table dans la base de données contenant tous les utilisateurs. Chaque ligne contient l'identifiant du sujet, son nom et son niveau de sécurité. Pour l'espèce des `objets`, il y a plusieurs possibilités. Nous avions tout d'abord songé à considérer une cellule d'une table comme un objet, mais cela imposait d'avoir un niveau de sécurité pour chaque cellule, ce qui est relativement coûteux en terme de mémoire. Nous avions également pensé prendre chaque ligne comme un objet, mais cela implique d'exécuter chaque requête SQL afin de savoir quelles sont les lignes concernées par cette requête, ce qui pose d'évidents problèmes de sécurité, comme par exemple dans le cas d'une requête `DELETE`. Finalement, nous avions le choix entre les colonnes ou les tables pour définir les objets, et pour des raisons de simplicité de traitement des requêtes SQL, nous avons choisis de définir un objet comme une table. De la même manière que pour les sujets, il existe une table contenant l'identifiant unique de chaque table, son nom et son niveau de sécurité. Il est intéressant de constater qu'étant donné que la table des sujets et celles des objets ne sont pas considérées comme des objets, il n'est pas possible pour les utilisateurs d'accèder à ces tables.

Les classifications sont fournies avec un ordre partiel (dans l'implémentation actuelle nous l'utilisons l'exemple classique : `public` $\leq$ `confidentiel` $\leq$ `secret`).

3.2.2. Architecture

Nous considérons qu'il existe une partie privée de la base de donnée seulement utilisable par le moniteur de référence (le programme Focal). Cette partie est utilisée pour stocker des informations comme les droits d'accès, les niveaux de sécurité, etc.

Quand le programme démarre, l'état courant de Bell & La Padula est calculé à partir des informations stockées dans la partie privée de la base de donnée. Ensuite, le programme appelle une fonction externe, faite en OCaml, qui «parse» les requêtes SQL vers un arbre de syntaxe abstraite et retourne la première requête Bell & La Padula à exécuter. Cette requête est exécutée et la réponse est retournée à la fonction externe. Selon la réponse, la fonction calcule une autre requête Bell & La Padula ou non. Quand chaque requête Bell & La Padula correspondant à la requête SQL a été exécutée et pour chacune d'elle, la réponse est toujours `yes`, alors la requête SQL est exécutée par le serveur MySQL et le résultat est retourné à l'utilisateur. Si une des réponses aux requêtes Bell & La Padula est `no`, alors un message d'erreur est retourné.

Quand le programme s'arrête, l'état courant de Bell & La Padula est stocké dans la partie privée de la base de donnée.

Ces opérations sont décrites dans la figure 3.

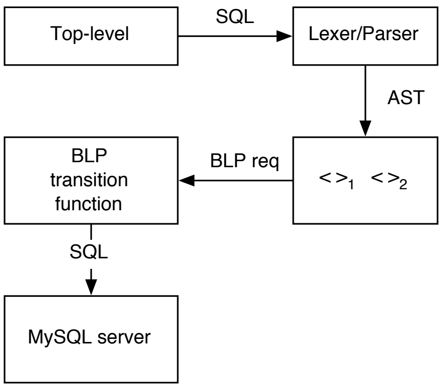

Figure 3: Architecture du programme

L'intégration avec MySQL se fait via OCaml qui bénéficie d'une interface avec le langage de requête relationnel. Le programme Focal fait donc appel à des fonctions OCaml pour accéder à la base de donnée.

Conclusion

Ce développement montre que les méthodes formelles peuvent être facilement utilisées pour accroître la sécurité d'un système. En effet, la formalisation en Focal de la fonction de transition de Bell & La Padula permet d'assurer (à travers les preuves) la sûreté du contrôle d'accès. Cela peut faciliter le processus de certification pour les Critères Communs par exemple.

Nous obtenons aussi une politique de sécurité réutilisable avec n'importe quel serveur SQL existant sans en modifier la structure. De plus, il est possible de définir facilement d'autres politiques de sécurité. Par exemple : la politique de sécurité de Biba [6] peut être obtenue en définissant le treillis des niveaux de sécurité de Bell & La Padula en son dual. Cette manipulation est relativement simple dans Focal grâce à la librairie de calcul formel. Enfin, cette formalisation nous a permis de distinguer trois modes d'accès possible en fonction du degré de sécurité voulu :le mode requête, facile à intégrer mais pouvant violer la propriété mac-$\star$ de Bell & La Padula par canal caché (l'écran par exemple), le mode strict où on ne relâche jamais les accès, et enfin le mode session où les accès sont empilés/dépilés. Ces deux derniers modes sont significatifs de deux aspects de la sécurité en informatique. Le premier est le compromis entre une sécurité absolue et une convivialité du système. Le deuxième est le fait que si l'on veut un système relativement convivial, alors il ne faut pas vouloir définir la non-pénétrabilité du système, mais simplement augmenter les temps et les moyens nécessaires à un attaquant pour pénétrer le système.

Notons qu'il ne s'agit pas ici de proposer une base de données sécurisée mais de proposer un module de sécurité pour les bases de données, garantissant ainsi la sécurité seulement si l'on passe par ce module. Un individu malicieux peut toujours pirater la base de donnée si il contourne le module.

Cette implantation a été testée sur quelques exemples mais il reste encore à réaliser une étude comparative afin de pouvoir estimer le surcoût par rapport à un serveur SQL classique avec et sans politique de sécurité.

Plusieurs optimisations sont possibles pour ce développement. La première est d'intégrer le contrôle d'accès dans le serveur. En effet, actuellement, chaque requête est traitée deux fois : une fois par notre module, et une fois de manière classique par le serveur. Cela implique que nous sommes en surcoût total.

La deuxième optimisation serait de limiter le nombre de requêtes Bell & La Padula exécutées. En effet, avec notre sémantique, plusieurs requêtes SQL différentes peuvent correspondre aux mêmes requêtes Bell & La Padula. Par exemple, si un utilisateur effectue plusieurs `SELECT` de suite sur la même table, il demande à chaque fois le même accès en lecture. Avec notre politique de sécurité et si nous supposons qu'il n'y a pas d'accès en parallèle, alors il suffit de faire une seule demande d'accès en lecture, puis un seul relâchement d'accès.

Enfin, nous pouvons implanter cette formalisation dans d'autres domaines que les bases de données comme l'application traditionnelle des politiques de sécurité dans la gestion des fichiers.

Remerciements

Nous remercions vivement Damien Doligez, Emmanuel Guréghian, Thérèse Hardin, Mathieu Jaume, Virgile Prevosto et Renaud Rioboo pour toutes les discussions fructueuses que nous avons eues à propos de ce travail.

Bibliographie

[1] D. Bell and L. LaPadula. Secure Computer Systems: a Mathematical Model. Technical Report MTR-2547 (Vol. II), MITRE Corp., Bedford , MA, May 1973.

[2] E. Gureghian, Th. Hardin, and M. Jaume. A full formalisation of the Bell and Lapadula security model. Technical Report 2003-007, Univ. Paris 6, LIP6, 2003.

[3] M. Jaume and C. Morisset. Formalisation and implementation of access control models. In *Information Assurance and Security (IAS'05) Inter national Conference on Information Technology, ITCC*, pages 703–708. IEE E CS Press, 2005.

[4] L.J. LaPadula and D.E. Bell. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4:239–263, 1996.

[5] McLean. The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, page s 2–7. IEEE Computer Society Press, 1988.

[6] K. Biba. Integrity consideration for secure computer systems. Technical Report MTR-3153, MITRE Corporation, 1975.

[7] Logical Project. *The Coq Proof Assistant Reference Manual Version 7*. INRIA-Rocquencourt, 2002.

[8] R. Rioboo, D. Doligez, V. Prevosto, M. Jaume, M. Maarek, C. Dubois, S . Fechter, V. Ménissier-Morain, O. Pons, D. Delahaye, V. Viguié, and T. Ha rdin. *FoC, version 0.0 Tutorial and reference manual*. LIP6 – INRIA – CNAM, jui 2003. Distribution available at: `http://focal.inria.fr`.

A. Syntaxe abstraite de SQL

Cette syntaxe abstraite est basée sur la syntaxe concrète de SQL définie dans le serveur MySQL 5.0 (*http://www.mysql.com*).

Une requête SQL peut avoir deux formes.

La forme simple:

```
SELECT * FROM t1, t2;

INSERT INTO t1 VALUES (2, 3, 4);
```

La forme imbriquée, où la requête dépend du résultat d'une ou plusieurs requêtes `SELECT`:

```
CREATE TABLE t1 SELECT name FROM t2;

INSERT INTO t1 SELECT name FROM t3;
```

Le *select-statement* est défini comme:

$$\begin{array}{rcl} sel ::= & & \perp \\ & | & SEL(o, sel) \end{array}$$

et la syntaxe abstraite pour les requêtes SQL :

$$\begin{array}{rcl} ast ::= & & sel \\ & | & CRE(o, sel) \\ & | & DEL(o, sel) \\ & | & INS(o, sel) \\ & | & UPD(o) \\ & | & SEL(o, sel) \\ & | & DRO(o) \end{array}$$

B. Sémantique de SQL

B.1. Règles d'inférence pour $<>_g$

B.1.1. $\perp$

$$(\perp)\frac{}{<\perp, \rho>_g \rightarrow (\mathsf{yes}, \rho)}$$

B.1.2. CREATE

$$(CRE_g^1)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}{< CRE(o, sel), s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}$$

$$(CRE_g^2)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{yes}, \rho') \quad t_C((\sigma_\epsilon, C, \sigma_\epsilon, o, \chi_f(o, c, E)), \rho') = (\mathsf{no}, \rho')}{< CRE(o, sel), s, c, \rho >_g \rightarrow (no, \rho)}$$

$$(CRE_g^3)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{yes}, \rho') \quad t_C((\sigma_\epsilon, C, \sigma_\epsilon, o, \chi_f(o, c, E)), \rho') = (\mathsf{yes}, \rho'')}{< CRE(o, sel), s, c, \rho >_g \rightarrow t_C((\sigma_\epsilon, C, \sigma(s), o, \chi_\epsilon), \rho'')}$$

B.1.3. INSERT

$$(INS_g^1)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}{< INS(o, sel), s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}$$

$$(INS_g^2)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{yes}, \rho')}{< INS(o, sel), s, c, \rho >_g \rightarrow t_G((\sigma_\epsilon, G, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{a})), \rho')}$$

B.1.4. DELETE

$$(DEL_g^1)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}{< DEL(o, sel), s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}$$

$$(DEL_g^2)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{yes}, \rho')}{< DEL(o, sel), s, c, \rho >_g \rightarrow t_G((\sigma_\epsilon, G, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{w})), \rho')}$$

B.1.5. UPDATE

$$(UPD_g)\frac{}{< UPD(o), s, c, \rho >_g \rightarrow t_G((\sigma_\epsilon, G, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{w})), \rho)}$$

B.1.6. SELECT

$$(SEL_g^1)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}{< SEL(o, sel), s, c, \rho >_g \rightarrow (\mathsf{no}, \rho)}$$

$$(SEL_g^2)\frac{< sel, s, c, \rho >_g \rightarrow (\mathsf{yes}, \rho')}{< SEL(o, sel), s, c, \rho >_g \rightarrow t_G((\sigma_\epsilon, G, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{r})), \rho')}$$

B.1.7. DROP

$$(DRO_g)\frac{}{< DRO(o), s, c, \rho >_g \rightarrow t_D((\sigma_\epsilon, D, \sigma(s), o, \chi_\epsilon), \rho)}$$

B.2. Règles d'inférence pour $<>_r$

B.2.1. INSERT

$$(INS_r^1)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}{< INS(o, sel), s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}$$

$$(INS_r^2)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{yes}, \rho')}{< INS(o, sel), s, c, \rho >_r \rightarrow t_R((\sigma_\epsilon, R, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{a})), \rho')}$$

B.2.2. DELETE

$$(DEL_r^1)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}{< DEL(o, sel), s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}$$

$$(DEL_r^2)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{yes}, \rho')}{< DEL(o, sel), s, c, \rho >_r \rightarrow t_R((\sigma_\epsilon, R, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{w})), \rho')}$$

B.2.3. UPDATE

$$(UPD_r)\frac{}{< UPD(o), s, c, \rho >_r \rightarrow t_R((\sigma_\epsilon, R, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{w})), s, c, \rho)}$$

B.2.4. SELECT

$$(SEL_r^1)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}{< SEL(o, sel), s, c, \rho >_r \rightarrow (\mathsf{no}, \rho)}$$

$$(SEL_r^2)\frac{< sel, s, c, \rho >_r \rightarrow (\mathsf{yes}, \rho')}{< SEL(o, sel), s, c, \rho >_r \rightarrow t_R((\sigma_\epsilon, R, \sigma(s), o, \chi_{\mathcal{A}}(\mathsf{r})), s, c, \rho')}$$

B.2.5. Autres requêtes

Pour toute autre requête Q (y compris $\perp$)

$$(OTH_r)\frac{}{< Q, s, c, \rho >_r \rightarrow (\mathsf{yes}, \rho)}$$