Utilisations de la mutation pour les tests de contrôle d'accès dans les applications

Tejeddine Mouelhi (tejeddine.mouelhi@telecom-bretagne.eu)*
Yves Le Traon (yves.letraon@telecom-bretagne.eu)*
Benoit Baudry (bbaudry@irisa.fr)†

Résumé : La sécurité des applications web est un enjeu majeur pour les entreprises et les organisations. Cela implique la mise en place de mécanismes de sécurité pour renforcer la protection de ces applications contre les attaques et garantir ainsi les différents objectifs de sécurité (confidentialité, intégrité et disponibilité). Parmi ces mécanismes de sécurité, le mécanisme de contrôle d'accès permet de garantir que l'accès des utilisateurs aux ressources respecte une certaine politique de contrôle d'accès. La validation du mécanisme de contrôle d'accès est importante pour garantir la sécurité de l'application et l'absence de failles. Pour cela, nous proposons une technique d'injection de fautes en se basant sur un modèle de fautes pour politiques de contrôle d'accès. L'injection de fautes (mutation) est utilisée aussi pour tester la robustesse du mécanisme de sécurité pour détecter les mécanismes de sécurité cachés. Nous avons appliqué notre approche à trois applications web pour obtenir des résultats expérimentaux et pour montrer l'efficacité de notre approche concernant la détection de mécanismes cachés.

Mots Clés : contrôle d'accès, test de politique de contrôle d'accès, analyse de mutation

1 Introduction

La sécurité des applications web est un enjeu majeur pour les entreprises et les organisations qui utilisent ces applications comme vitrines commerciales ou moyen principal de communication. L'amélioration de la sécurité implique l'obligation de spécifier et de mettre en place des techniques de sécurité pour renforcer la protection de ces applications contre les attaques et garantir ainsi les différents objectifs de sécurité (confidentialité, intégrité et disponibilité). Parmi ces techniques de sécurité, le contrôle d'accès (CA) permet de garantir que l'accès des utilisateurs aux ressources respecte une certaine politique de sécurité bien définie. Les experts en sécurité, en se basant sur un modèle de CA, définissent les exigences de sécurité, sous forme d'un ensemble de règles de CA qui régissent l'accès des utilisateurs aux ressources en autorisant et interdisant l'accès suivant le rôle de l'utilisateur et le contexte (spatial, temporel, etc.). Ce modèle est ensuite implanté dans l'application via un mécanisme de sécurité, et ce utilisant une des architectures disponibles (comme par exemple XACML, JAAS ou EJB) ou en développant une solution ad-hoc. Il est évident que le mécanisme de CA doit absolument être conforme à la politique de CA qu'il implante, pour qu'il n'autorise pas un accès interdit.

∗. Telecom Bretagne (projet Serval, RSM) 2, rue de la Châtaigneraie CS 17607 35576 Cesson Sévigné Cedex, France
†. IRISA/INRIA 35042 Rennes, France

Un rapport récent [CWE09], produit d'une collaboration des principaux acteurs américains de la sécurité, dont entre autres de grands éditeurs de logiciel (Microsoft, Symantec), des entreprises d'audit de sécurité d'applications, classe dans les 25 erreurs de programmation les plus néfastes sur la sécurité, ces problèmes liés à la non-conformité ou à la mauvaise implantation de la politique de CA.

Pour garantir la bonne conformité du code implantant la sécurité par rapport à la politique de sécurité, nous avons entrepris d'étudier l'application de techniques de test classiques basées sur l'analyse de mutation. L'analyse de mutation est une technique de test qui permet d'évaluer la qualité des tests et leur capacité à détecter les erreurs. Elle permet aussi de comparer les critères et les stratégies de génération des tests. Cette technique est basée sur l'injection systématique de fautes dans le système. Nous avons adapté cette technique [TMB07] pour tester l'implantation des politiques de CA, étudier et proposer des solutions pour le test des mécanismes de CA, la robustesse de ces derniers, ainsi que la détection des mécanismes de CA cachés. Ces trois études seront détaillées par la suite, respectivement dans les sections 4,5 et 6.

La première étude [TMB07] s'est focalisée sur la problématique de test du mécanisme de CA. En partant du constat que les tests fonctionnels en exécutant certains scénarii déclenchent le mécanisme de CA, nous avons étudié si les tests fonctionnels sont suffisants pour tester le mécanisme de CA. En utilisant la mutation, nous avons donc comparé les capacités de détection des erreurs par les tests fonctionnels et par les tests de sécurité. Nous avons proposé plusieurs critères et stratégies pour générer ces tests de sécurité. L'efficacité des critères a été comparée en utilisant la mutation. Cette étude sera détaillée dans la section 4.

La deuxième étude [TMB07] s'est penchée sur la robustesse des mécanismes de CA. L'objectif était d'étudier comment le comportement du mécanisme de CA peut être validé en utilisant la mutation. Ce comportement correspond aux règles de CA non spécifiées et pour lesquelles une politique par défaut s'applique (une permission ou une interdiction). Il revient donc aux testeurs de bien vérifier que la politique par défaut s'applique bien dans tous les cas. L'analyse de mutation a ainsi été utilisée dans ce but. La méthode employée sera décrite dans la section 5.

La troisième étude [TMPB08] a abordé un problème intrinsèque aux systèmes existants (connu en anglais sous le nom 'legacy systems') qui ont été développés en embarquant du code implantant une politique de CA. Si la documentation sur ce code n'est plus disponible ou si le langage de développement utilisé est difficile à maintenir (comme Cobol par exemple), l'évolutivité de la politique de CA pose problème. En effet des règles de CA peuvent être implantées par des mécanismes qui ne sont plus contrôlables (faute de traçabilité). Ces mécanismes cachés doivent être détectés quand on veut faire évoluer ou contrôler la politique de CA. Il est possible d'employer l'audit manuel de code pour trouver ces mécanismes cachés. Mais cette tache est souvent fastidieuse voire très difficile quand le code est impossible à maintenir (très peu commenté). Une solution assistée s'avère donc indispensable. C'est pour cela que nous proposons d'utiliser l'analyse de mutation dans le but de détecter ces mécanismes cachés.

Les résultats présentés dans cet article sont dans la plupart des résultats d'études empiriques menées sur 3 systèmes d'applications web. La suite se répartit comme suit. La section 2 définira les concepts de sécurité et de tests utilisés dans cet article. La section 3, présentera l'analyse de mutation adaptée aux tests de mécanismes de CA. Par la suite,

les sections 4, 5 et 6 détailleront les 3 études menées. Enfin, avant de conclure, la section 7 dressera un état de l'art des travaux menés sur le test de mécanisme de CA.

2 Contexte

Les principaux concepts utilisés dans la suite sont présentés dans cette section. Il s'agit des concepts de sécurité ; les modèles de CA et l'architecture de sécurité implantant le CA, mais aussi des concepts tests ; l'analyse de mutation, les tests fonctionnels et les tests de sécurité.

2.1 Le contrôle d'accès

Les modèles de CA (comme OrBAC [KBB+03] ou RBAC [FSG+01] par exemple) permettent de définir un ensemble de règles qui constitueront la politique de CA. Dans cette étude, nous considérons un modèle inspiré par RBAC et étendu pour intégrer les notions de contexte et les règles d'interdiction. Nous avons essayé d'avoir une approche plus générale qui ne soit pas limitée à un modèle donné. Plus formellement, une règle de CA consiste en cinq paramètres :

- *Statut S* : permission ou interdiction
- *Rôle R* : dans un domaine de noms défini RN
- *Permission P* : dans PN
- *Contexte C* : dans CN

Une règle de CA est définie ainsi : *S(R, P, C)*

Pour illustrer les règles d'une politique de CA, prenons l'exemple d'un système de librairie LMS (qui sera aussi utilisé après pour illustrer d'autres aspects) qui offre des fonctionnalités de gestion des livres, des comptes des utilisateurs et des prêts. Dans ce système, les utilisateurs peuvent réaliser trois opérations emprunter les livres, les réserver et les rendre. Les ressources dont on veut contrôler l'accès sont les livres et les comptes. Les entités (RN, PN) peuvent être ordonnées de manière hiérarchique. Par exemple, dans le cas de LMS, il y a deux types de rôles qui sont les emprunteurs et le personnel. Les rôles Etudiant et Enseignant héritent donc du rôle Emprunteur. Cette hiérarchisation permet de définir des règles au niveau du rôle Emprunteur pour qu'elles soient ensuite appliquées pour les rôles Etudiant et Enseignant. Enfin, on distingue trois contextes temporels qui sont les jours travaillés, les jours fériés et les jours de maintenance. Les différentes entités sont présentées dans la figure 1.

Une fois les entités définies, il faut écrire la politique de contrôle d'accès. Voici quelques exemples de règles :

- *R1 : Permission(Administrator,CreateAccount,default)*
- *R2 : Permission(Borrower,BorrowerActivities,WorkingDays)*
- *R3 : Prohibition(Borrower,ModifyAccount,default)*

On distingue 2 types de règles : les règles *primaires* et les règles *concrètes*. Les règles primaires représentent l'ensemble des règles définies. Quant aux règles concrètes, ce sont l'ensemble des règles obtenues après avoir dérivé les règles primaires en se basant sur la hiérarchie. C'est le cas de R2, qui s'applique à 'BorrowerActivity' et qui désigne l'ensemble des actions qu'un emprunteur peut effectuer (qui sont d'après la figure 1 : *BorrowerBook*, *ReserveBook* et *ReturnBook*). Les règles primaires n'incluent pas ces règles dérivées.

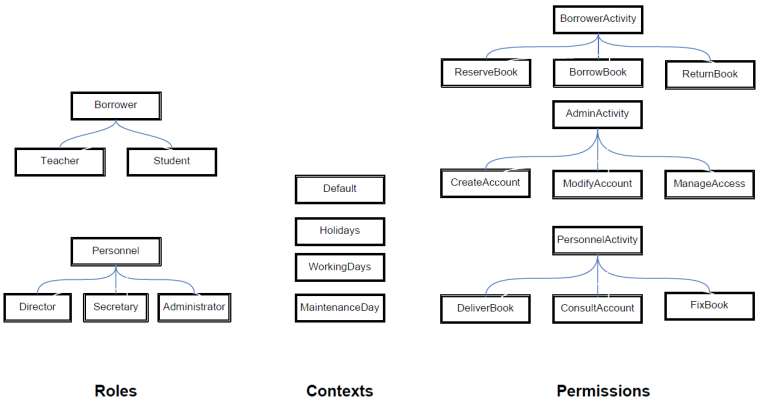

Figure 1: Les entités de LMS

La distinction entre règles primaires et concrètes est importante pour la suite de l'article, quand on définira les critères de génération de tests. Après avoir défini la politique de CA, l'étape suivante consiste à implanter cette politique dans le système à développer. Le test doit être naturellement adapté à cette architecture de déploiement.

2.2 Architecture d'intégration de la politique de contrôle d'accès et tests

Une architecture typique d'une application implantant une politique de sécurité, comme présenté dans la figure 2, doit séparer la partie logique métier (en anglais " Business Logic ") de la partie CA. La partie *logique métier* contient le code implantant des exigences fonctionnelles du système. Dans LMS, elle contient, par exemple, les modules permettant d'emprunter un livre, de mettre à jour la base de données (s'il y en a) et de modifier les comptes etc. La politique de CA est encapsulée dans un composant appelé PDP (pour *Policy Decision Point*). Ce dernier stocke la politique de CA (soit dans un fichier XACML, un fichier texte, une BD etc.) et permet de configurer la politique pour mettre à jour les règles. Le PDP est utilisé par les PEPs (pour *Policy Enforcement Point*). Ces derniers sont placés dans les points de l'application où vont s'appliquer les règles de sécurité. Généralement, ils sont placés avant l'exécution des fonctionnalités à contrôler par la politique de CA. Les PEP envoient des requêtes avec les informations sur le rôle de l'utilisateur, l'activité et la vue demandés, ainsi que le contexte (à calculer souvent) au PDP qui décide si l'accès demandé est autorisé ou interdit en se basant sur la politique de CA.

Le PDP peut être généré automatiquement à partir d'une politique de CA [MFBT08a]. Quant au PEP, il faudra l'implanter d'une manière Ad-hoc, manuellement pour bien s'adapter au code et à l'architecture de l'application. Il revient donc aux testeurs de bien valider que le PDP est correctement appelé par le PEP et ainsi que les règles de CA s'appliquent correctement. L'objectif est de garantir la bonne intégration de la politique de CA dans la partie logique métier en utilisant les tests.

2.3 L'analyse de mutation

L'analyse de mutation sert à évaluer les tests. Elle consiste à injecter des fautes dans un programme pour en créer des versions erronées qu'on appelle *mutants*. Un mutant est une

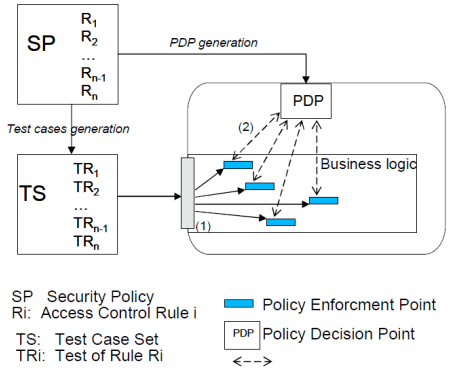

Figure 2: Test d'une architecture type implantant une politique de contrôle d'accès

version du programme initial contenant une erreur. Les tests sont lancés sur les mutants pour détecter les erreurs injectées. Cela garantit que les tests sont bien capables de détecter d'éventuelles erreurs, et permet d'avoir une unité de mesure intéressante de l'efficacité des tests. Cette mesure est appelée le *score de mutation*, qui est le pourcentage des mutants détectés (dits tués) par rapport aux nombre total de mutants. Le but est naturellement d'atteindre un score de mutation de 100%. Cependant, il arrive que les mutants créés se comportent exactement comme la version initiale du programme. Ce sont des mutants équivalents. Il est donc important de pouvoir détecter les mutants équivalents pour avoir un score de mutation significatif (et non pas sous-évalué). Il est à noter que la génération de mutants peut être automatisée par des opérateurs de mutations. Chaque opérateur injecte un type particulier d'erreur. Dans le cas d'un programme (par exemple en C ou en Java), des opérateurs de mutation classiques consistent à permuter les opérateurs relationnels ou logiques. Pour chaque type de langage (par exemple pour les langages orientés objet), des jeux d'opérateurs existent qui reflètent les erreurs classiques de programmation.

3 La mutation pour le test des politiques de contrôle d'accès

Nous nous sommes inspirés de l'analyse de mutation au niveau du code pour la transposer au CA. Dans cette section, nous présentons l'approche adoptée ainsi que les résultats en terme de nombre de mutants générés pour les cas d'étude.

3.1 L'approche pour adapter la mutation

Plusieurs politiques mutantes sont d'abord créées à partir de la politique de CA initiale. Elles sont ensuite implantées pour avoir plusieurs versions de l'application, exécutant chacune une politique de CA erronée (figure 3). Cela est rendu possible par le fait que la politique est encapsulée dans le PDP. On procède donc au remplacement de la politique utilisée par le PDP pour obtenir une implantation de la politique mutante. Ensuite, les tests sont générés à partir de la politique de CA initiale pour être lancés sur les mutants pour détecter les erreurs injectées précédemment, et ainsi mesurer le score de mutation qui correspond au pourcentage de mutants tués par rapport au nombre total de mutants.

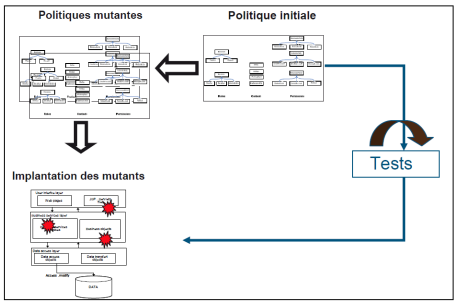

Figure 3: La Mutation appliquée aux politiques de contrôle d'accès

La création des mutants est effectuée de manière systématique via des opérateurs de mutation. Chaque opérateur injecte un type particulier d'erreur. Une seule erreur est injectée à la fois, pour créer chaque mutant. Nous avons proposé les opérateurs de mutations suivants :

- Opérateurs basiques modifiant le type :
  - PPR (permission to prohibition) : remplace une règle de permission par une interdiction.
  - PRP (prohibition to permission) : remplace une règle d'interdiction par une permission.
- Opérateurs basiques modifiant les paramètres :
  - RRD (role replaced with different one) : remplace dans une règle le rôle d'une règle par un autre rôle pris au hasard.
  - CRD (context replaced with different one) : remplace dans une règle le contexte par un autre contexte pris au hasard.
- Opérateurs basiques modifiant la hiérarchie :
  - RPD (parent role replaced with a descendant) : remplace dans une règle un rôle par un de ses descendants (modifiant ainsi les règles dérivées)
  - APD (parent action replaced with a descendant) : remplace dans une règle une permission par un de ses descendants (modifiant les règles dérivées)
- Opérateur avancé :
  - ANR : ajoute une nouvelle règle ne faisant pas partie des règles définies.

Il est important de souligner le fait que ces opérateurs ne produisent pas de mutants équivalents. En effet, les mutants créés contiennent par construction des règles différentes de la politique initiale parce qu'ils ajoutent une nouvelle règle ou modifient une règle existante.

On distingue deux types d'opérateurs de mutation ; les opérateurs basiques (tous les opérateurs sauf ANR) et l'opérateur avancé de mutation qui est ANR. Ce dernier est particulier parce qu'il vise à tester le comportement par défaut du mécanisme de CA. En effet, toute politique de CA contient X règles et 1 règle par défaut (permission ou interdiction) qui s'appliquent quand les autres règles ne correspondent pas aux entités d'une requête. L'opérateur ANR permet de créer des règles qui viennent se substituer à la règle par défaut et viennent compléter les X règles définies. Il s'agit donc d'un opérateur avancé puisqu'il permet de tester plutôt la robustesse du mécanisme de CA

3.2 La mutation appliquée aux cas d'étude

Dans notre étude, nous avons utilisé les trois cas d'étude suivants :
- LMS : système de gestion de bibliothèque.
- VMS : système de gestion de réunion virtuelle.
- ASMS : système de gestion de vente aux enchères.

Quelques informations sur la taille de ces applications sont montrées dans le tableau suivant :

	Nombre de classes	**Nombre de méthodes**	**Lignes de code**
LMS	62	335	3204
VMS	134	581	6077
ASMS	122	797	10733

Ces trois applications implantent des politiques de CA. Voici le nombre de règle pour chaque système :

	Règles
LMS	41
VMS	106
ASMS	130

Le tableau suivant présente les mutants générés pour chaque système. On peut noter que l'opérateur ANR génère un nombre important de mutant. Cela s'explique par le fait qu'il ajoute à chaque fois une nouvelle règle non définie par la politique initiale en combinant au hasard les différentes entités. Les mutants générés sont donc nombreux quand la politique définit peu de règles et contient un nombre conséquent d'entités (rôle, permission et contextes).

Catégorie des opérateurs		**Opérateur**	**LMS**	**ASMS**	**VMS**
Opérateurs basiques	Modificateur de type	PPR	22	89	36
		PRP	19	41	70
	Modificateur de paramètre	RRD	60	650	530
		CRD	60	520	318
	Modificateur d'hiérarchie	RPD	5	20	20
		APD	5	0	20
Opérateur avancé		ANR	200	736	432
Total			**371**	**2056**	**1426**

4 Tests du mécanisme de contrôle d'accès

Les tests de sécurité sont générés à partir de la politique de CA. Ils ont pour objectif de valider la conformité du mécanisme de sécurité vis-à-vis de sa politique de CA. En pratique, la politique de CA est souvent connectée aux fonctionnalités du système. Tester les fonctionnalités du système permet donc d'activer et ainsi de valider la conformité à la politique de CA. Nous abordons les deux questions suivantes :
- Est-ce que tester les fonctionnalités permet de valider le mécanisme de sécurité ?
- Comment compléter ces tests avec des tests spécifiques à la sécurité ?

Tout test, qu'il soit de sécurité ou fonctionnel est composé de trois parties : l'intention du test (ce qu'on veut tester), la séquence du test (la suite d'opérations à effectuer) et l'oracle (qui vérifie et décide si le test réussi ou échoue). Pour illustrer la différence entre les tests fonctionnels et les tests de sécurité, nous présentons ces deux exemples simples :

Test fonctionnel : Test qu'un emprunteur emprunte et retourne un livre
- *Intention* : tester l'emprunt et le retour d'un livre par un emprunteur
- *Séquence* : emprunter un livre et ensuite le rendre pendant les jours travaillés

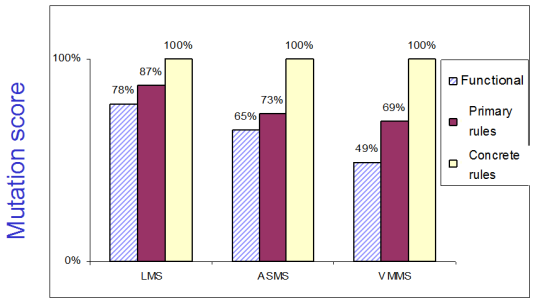

Figure 4: Résultats empiriques de la mutation

– *Oracle* : livre disponible à nouveau et BD mise à jour.

Test de sécurité : Test qu'un emprunteur peut emprunter et rendre un livre les jours travaillés (comme spécifié par la politique de CA) :

– *Intention* : tester qu'un emprunteur a le droit d'emprunter et rendre un livre les jours travaillés
– *Séquence* : emprunter un livre et ensuite le rendre pendant les jours travaillés
– *Oracle* : Interroger le PDP pour vérifier que les bonnes règles ont été activées.

Les tests de sécurité peuvent être générés à partir de différents critères. Nous proposons les critères suivants :

– *Toutes les règles primaires* : couvrir les règles primaires, si une règle est dérivable, tester une de ses règles dérivées.
– *Toutes les règles concrètes* : couvrir l'ensemble des règles concrètes.

Pour évaluer la qualité des tests fonctionnels et la comparer avec celle des tests générés à partir de ces deux critères, nous avons lancé les tests sur les mutants basiques. Les tests fonctionnels couvrent le code de l'application.

La figure 4 présente les résultats obtenus. On remarque clairement que les tests fonctionnels n'atteignent pas un score de 100% et sont même très insuffisants pour VMS (score de mutation de 49% seulement). Quant aux tests de sécurité, ils obtiennent de meilleurs résultats et les tests basés sur le critère " *Toutes les règles concrètes* " atteignent logiquement le score de 100%.

4.1 Tests de la robustesse du mécanisme de sécurité

Dans cette section, nous étudions la capacité des tests générés à partir des critères à valider le comportement par défaut et donc à tester la robustesse du mécanisme de sécurité. Nous les comparons à un nouveau critère présenté ci-après et qui est bien adapté pour tester le comportement par défaut :

Toutes les règles par défaut (critère 3) : ce critère a pour but de valider toutes les règles par défaut non spécifiées dans la politique de sécurité.

On veut aussi étudier si les tests générés à partir du critère 3 sont suffisamment performants pour couvrir la politique de CA et donc s'ils sont capables de détecter les mutants basiques

Le tableau suivant présente les résultats obtenus. Nous constatons que les tests du critère " *Toutes les règles par défaut* " ne sont pas suffisant pour couvrir la politique de

CA puisqu'ils ne détectent pas suffisamment de mutants basiques. Et d'un autre coté, les tests basés sur le critère " *Toutes les règles concrètes* " ne couvrent qu'un sous ensemble limité des mutants ANR. Ils ne sont pas donc suffisants pour tester la politique par défaut.

		Nombres de tests	Mutants basiques	Mutants ANR
LMS	Toutes les règles concrètes	35	100%	17%
	Toutes les règles par défaut	154	59%	100%
ASMS	Toutes les règles concrètes	110	100%	16%
	Toutes les règles par défaut	614	69%	100%
VMS	Toutes les règles concrètes	106	100%	32%
	Toutes les règles par défaut	384	72%	100%

Pour résumer, la bonne stratégie consiste à utiliser des tests bien spécifiques à la sécurité et à combiner les deux critères Couvrir *toutes les règles concrètes* et Couvrir *toutes les règles par défaut*. La mutation servira de support efficace pour mesurer la qualité des tests.

5 La mutation pour détecter les mécanismes cachés

Nous abordons, dans cette partie, l'utilisation de la mutation comme moyen efficace pour détecter les mécanismes cachés. On décrit d'abord le problème à résoudre pour ensuite présenter l'approche qui est basée sur la mutation. Nous présentons également une démarche générale fondée sur la mutation pour guider l'évolution de la politique de CA d'un système. Finalement, nous présentons quelques résultats expérimentaux obtenus en appliquant l'approche aux trois cas d'étude.

5.1 Les mécanismes cachés de contrôle d'accès

La meilleure approche pour implanter une politique de CA consiste à séparer le PDP de la partie logique métier. Cependant, la séparation est rarement parfaitement nette, surtout dans les systèmes anciens. Dès lors qu'il s'agit de faire évoluer la politique de contrôle d'accès de ces applications, il est nécessaire de pouvoir localiser ces endroits du code empêchant l'évolution du système parce qu'ils implantent directement des règles de CA dans le code. Les mécanismes de contrôle d'accès peuvent être de natures différentes. Ainsi on distingue les mécanismes *explicite visibles, explicites cachés* et *implicites.*

Les *mécanismes explicites* sont implantés dans du code. Les mécanismes explicites sont visibles quand on dispose d'un lien de traçabilité ou d'une documentation permettant d'établir une relation avec la politique de CA (ou une partie de celle-ci). Un mécanisme explicite est dit caché quand on ne dispose pas d'information permettant de lier ce mécanisme à une règle de la politique de CA. L'exemple de la figure 5 illustre deux exemples typiques de mécanismes explicites (un visible et un caché). Le premier fait appel à un composant externe (qui est le 'securityPolicyService') en charge d'appliquer la politique de sécurité, un composant contrôlable qui est basé sur une politique de contrôle d'accès. Quant au deuxième, il s'agit d'un mécanisme caché puisqu'il applique directement une règle de contrôle d'accès (interdisant l'accès en se basant sur une condition). Souvent, dans les systèmes anciens (développés notamment en Cobol), plusieurs mécanismes similaires peuvent se trouver dans le code de l'application. La documentation du code (modèle, commentaire etc.) est parfois absente, et la maitrise du code est ardue. En plus, le risque avec les mécanismes explicites cachés est qu'ils contournent parfois les nouvelles règles de CA, et peuvent ainsi servir de " back-door ".

```
public void borrowBook(Book b, User user) {
// visible mechanism, call to the security policy service
SecurityPolicyService.check(user,
SecurityModel.BORROW_METHOD, Book.class, SecurityModel.DEFAULT_CONTEXT);
// do something else
// hidden mechanism
If(getDayOfWeek().equals("Sunday") || getDayOfWeek().equals("Saturday")) {
// this is not authorized throw a business exception
throw new BusinessException("Not allowed to borrow in week-ends");}
...}
```

Figure 5: Exemple de mécanismes explicites de contrôle d'accès

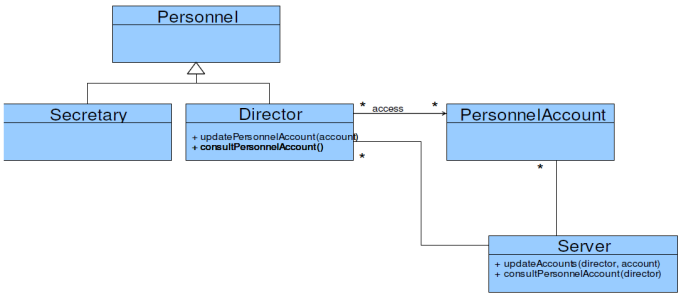

Figure 6: Exemple de mécanisme de contrôle d'accès implicite

Les *mécanismes implicites* sont imposés par l'architecture du code ou du système. Des contraintes sont donc exprimées par construction. Un exemple de mécanisme implicite est illustré dans le schéma de la figure 6. Dans cet exemple, par construction la classe Secrétaire ne dispose pas d'une méthode pour créer un compte. On remarque l'implantation implicite d'une règle de contrôle d'accès qui interdit aux secrétaires de créer les comptes. La politique de contrôle d'accès est exprimée implicitement via le modèle de classe, ce qui rend son évolution plus délicate. Dans cet exemple, le fait de permettre aux secrétaires de mettre à jour les comptes revient, par exemple à ajouter dans la classe Serveur une nouvelle méthode " updateAccount(secretary,account) ". Dans ce cas la modification est facile à réaliser. Mais dans d'autres cas, la modification peut s'avérer difficile et fastidieuse voire impossible (surtout quand il y a un couplage important entre les classes).

Un mécanisme implicite est donc une contrainte de CA qui est intégrée dans l'architecture même du logiciel. Modifier une règle du CA implicitement implantée par un de ces mécanismes implique de faire du " refactoring " (réusinage) de l'architecture.

Qu'ils soient explicites ou implicites, les mécanismes de CA doivent être pris en compte lors de l'évolution de la politique de CA en suivant la démarche suivante :

1. Les mécanismes explicites visibles doivent être modifiés et testés
2. Les mécanismes explicites cachés peuvent être en conflit avec la nouvelle politique. Ils doivent donc être localisés et supprimés.
3. Les mécanismes implicites peuvent empêcher l'évolution de la politique. Dans ce cas le modèle doit être modifié voire remodelé pour rendre l'application plus flexible.

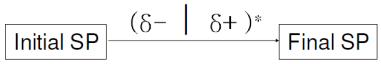

Figure 7: Les types de micro-évolutions

5.2 La mutation pour détecter les mécanismes cachés

Nous appliquons la mutation sur la politique de CA initiale en utilisant uniquement les deux opérateurs de modification de types PRP et PPR (permission vers interdiction et interdiction vers permissions). Cette étape permet de simuler une évolution incrémentale de la politique. Ensuite, les tests sont réalisés à partir de la politique mutante. Avant de lancer les tests, les mécanismes de sécurité visibles doivent être désactivés ce qui est possible puisque ils sont par nature localisables et les règles qu'ils activent sont connues. Si l'application est dépourvue de mécanismes cachés, toutes les requêtes de CA doivent être acceptées (il n'y a aucune interdiction). Les tests qui échouent indiquent donc forcément l'existence de mécanismes de sécurité cachés.

Cette approche dirigée par les tests permet de comprendre les mécanismes qui empêchent l'évolution de la politique de CA. Ils exécutent des scénarii, qui mettent en évidence les règles de CA implantées par des mécanismes cachés. Les tests permettent ainsi de détecter les mécanismes cachés et les règles implantées par ces derniers. Cette approche permet ainsi d'estimer *la flexibilité* du système. La flexibilité correspond au nombre de règles modifiables divisé par le nombre total de règles. Les règles sont modifiables quand elles ne sont pas implantées par des mécanismes cachés.

5.3 Guider l'évolution de la politique pas les tests et la mutation

La difficulté pour implanter une politique évolutive vient surtout du fait que les changements à faire sur le code ou le modèle de l'application sont très coûteux. En effet, cette modification nécessite l'audit du code de la partie logique métier. L'utilisation des tests offre donc une solution pragmatique à ce problème Il permet d'estimer ainsi le coût de l'évolution de la politique de CA. La démarche comprend les étapes suivantes :

1. Evaluer le système existant
2. Mesurer sa flexibilité (micro-evolutions possibles).
3. Diagnostiquer pour trouver les fonctions et ressources affectant la flexibilité.

Comme présenté dans la figure 7, il y a deux types de micro-évolutions :

1. $\delta+$: qui relâche la politique en ajoutant une permission ou en supprimant une interdiction.
2. δ- : qui restreint la politique en ajoutant une interdiction ou en supprimant une permission.

Soit PS une politique, donc un ensemble de permission et d'interdiction. Chaque micro-évolution génère une nouvelle politique. Notons PS- la politique résultant de δ- (une restriction) et PS+ la politique résultant de $\delta+$ (un relâchement).

La figure 8 présente l'approche dirigée par les tests qui permet d'estimer l'évolutivité de la politique :

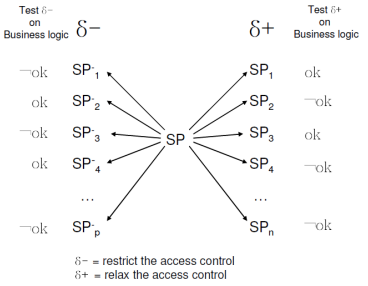

Figure 8: L'approche dirigée par les tests

1. On construit les différentes micro-évolutions
2. Pour chaque micro-évolution, le mécanisme de CA visible est désactivé
3. Pour chaque micro-évolution, un test est généré et lancé (sur la partie logique métier).
4. Si le test échoue cela indique que le système ne supporte pas cette micro-évolution. Une étape d'analyse et de modification éventuelle du code est nécessaire.

5.4 L'approche adoptée en pratique

La mesure de flexibilité qu'on propose, donne une indication sur toutes les micro-évolutions que le système tolère et qui ne nécessitent pas de modification ou remodelage de l'application. En pratique, il n'est pas nécessaire d'effectuer cette analyse complète.

Soit SPInit la politique initiale et SPTarget la nouvelle politique de CA. L'évolution à effectuer est exprimée par la formule suivante :

$$\Delta(SPinit) = SPTar = \delta_n^{-+} \circ \delta_{n-1}^{-+} \circ \delta_{n-2}^{-+} .... \delta_1^{-+}(SPinit)$$

En pratique l'approche à adopter est la suivante :

1. Ecrire un cas de test associé à chaque micro-évolution
2. Détecter les mécanismes cachés potentiels.
3. Fixer le problème et vérifier en ré-exécutant le test.
4. Implanter et documenter la micro-évolution dans le mécanisme visible.

5.5 Les cas d'étude

L'approche a été appliquée aux trois applications. Les différents résultats obtenus sont présentés plus en détail dans un article précédent [TMPB08]. Nous présentons ici les résultats du système VMS.

Les tableaux ci-après présentent les résultats décrivant la flexibilité pour chaque ressource et fonction. Ces fonctions sont les principaux points d'entrée du système. La flexibilité globale du système est mesurée à 0.35. Cela signifie que 35% des règles de la politique sont modifiables et donc qu'on peut appliquer des micro-évolutions liées à ces règles. Les résultats présentent aussi des informations plus détaillées sur :

1. Les ressources rigides et flexibles du système
2. Les fonctionnalités rigides et flexibles du système

Les tableaux montrent que les ressources PersonnelAccout et UserAccount sont flexibles et que c'est la ressource Meeting qui est la plus rigide et posera donc problème lors de l'évolution de la politique.

Enfin, on peut aller encore plus loin dans l'analyse et détailler pour chaque fonction son degré de flexibilité. Cela sert à déterminer les endroits du code abritant les mécanismes cachés.

Flexibilité globale

	Règles flexibles	Règles rigides	Flexibilité du système
Résultats	20	36	0,35

Flexibilité par ressource

Ressource	Règles flexibles	Règles rigides	Total des règles	Flexibilité
Meeting	12	36	48	0,25
PersonnelAccount	6	0	6	1
UserAccount	2	0	2	1

Flexibilité par fonction

Fonctionnalité	Flexibilité
updatePersonnelAccount	1
updateUserAccount	1
askToSpeak	0 ,13
leaveMeeting	0,14
overSpeak	1
closeMeeting	1
setMeetingAgenda	0,14
setMeetingModerator	0,14
speakInMeeting	0,14
setMeetingTitle	0,14
deleteUserAccount	1
openMeeting	1
handOver	1
deletePersonnelAccount	1

6 Travaux connexes

Récemment plusieurs travaux ont étudié le test des politiques de sécurité implantées dans les applications. On peut dans ce cadre citer les travaux de Xie et al. [HMHX07, MX07b] et de Groz et al. [DRG08, LMG07].

La mutation des politiques de CA a été proposée dans d'autres travaux de recherche par Xie et al. [MX07a] et appliquée aux tests des politiques basées sur XACML [LXVS05]. La mutation a été définie et utilisée dans le but d'évaluer plusieurs critères de génération automatiques de requêtes XACML. Ces requêtes servent à tester uniquement le PDP avec la politique implantée en XACML. Le test dans ce cas se limite à valider le PDP. Dans notre approche, nous considérons l'ensemble du mécanisme de sécurité en plus de l'application. Les tests que nous effectuons activent la politique de CA et testent que l'application applique bien la politique de CA.

L'approche appliquée par Xie et al. est limitée par le fait que la mutation est appliquée au niveau du code XACML (de manière syntaxique sur le code). Cela a pour effet de créer des mutants équivalents et la complexité du langage XACML rend difficile la détection de ces mutants équivalents. Les scores de mutation obtenus pour évaluer les stratégies de génération des tests sont donc forcément sous-estimés.

Plus généralement, la mutation a été appliquée à la sécurité [GOM98, DM00]. On peut citer par exemple, les travaux de Mathur et al [DM00] qui a appliqué la mutation pour perturber l'environnement de l'application qui contient les différents variables d'environnement système, les fichiers et les ressources dont dépend l'application. Cette perturbation a pour objectif d'observer le comportement de l'application dans un environnement perturbé et ainsi de vérifier que l'application agit de manière sécurisée. Cela signifie, qu'elle n'autorise pas d'accès interdits ou ne divulgue pas d'informations confidentielles

7 Conclusion

Nous avons présenté une synthèse d'un ensemble de travaux liés aux tests des mécanismes implantant les politiques de CA dans les applications. La technique de l'analyse de mutation a été instrumentée et a servi de support pour qualifier les tests de sécurité et pour évaluer les critères de génération des tests. Nous avons aussi abordé le problème des mécanismes de CA cachés. Nous avons proposé une approche basée sur les tests et la mutation pour résoudre ce problème. Les différentes approches ont été validées de manière expérimentale.

Par ailleurs, nous avons utilisé l'analyse de mutation des politiques de CA dans d'autres contextes qui n'ont pas été détaillés dans cet article faute de place. En effet, nous avons appliqué la mutation pour comparer différents critères de génération de cibles de tests à partir de la technique de test appelée " pair-wise " [PMT08].

Nous avons également proposé d'utiliser l'analyse mutation de manière générique, et ce indépendamment des modèles de CA. Pour cela, nous avons employé des techniques d'ingénierie dirigée par les modèles et nous avons proposé un méta-modèle pour les politiques de CA, qui a été utilisé pour modéliser des politiques en RBAC, OrBAC [MBF08], DAC et MAC [MFBT08b].

En se basant sur notre méta-modèle, et dans une approche dirigée par les tests, nous avons proposé une méthode de développement et plusieurs outils pour spécifier, déployer et tester les politiques de CA dans les applications [HMHX07]. Ce travail part de l'idée qu'il vaut mieux construire dès le départ, des applications facilement testables et pour lesquels on dispose de moyens efficaces de test (en utilisant la mutation) que d'avoir à gérer les problèmes de test de la politique de CA en aval.

Références

[CWE09] CWE/SANS TOP 25 Most Dangerous Programming Errors, 2009.

[DM00] Wenliang Du and Aditya P. Mathur. Testing for Software Vulnerability Using Environment Perturbation. In *International Conference on Dependable Systems and Networks*, pages 603 – 612, 2000.

[DRG08] V. Darmaillacq, J.-L. Richier, and R. Groz. Test generation and execution for security rules in temporal logic. In *IEEE International Conference on*

Software Testing Verification and Validation Workshop. ICSTW '08., pages 252–259, 2008.

[FSG+01] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3), 2001.

[GOM98] A. Ghosh, T. OĆonnor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software, 1998.

[HMHX07] Vincent C. Hu, E. Martin, J. Hwang, and T. Xie. Conformance Checking of Access Control Policies Specified in XACML. In *Proceedings of the 1st IEEE International Workshop on Security in Software Engineering*, 2007.

[KBB+03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control, 2003.

[LMG07] K. Li, L. Mounier, and R. Groz. Test Generation from Security Policies Specified in Or-BAC. In *31st Annual International Computer Software and Applications Conference. COMPSAC 2007.*, pages 255–260, 2007.

[LXVS05] eXtensible Access Control Markup Language and http ://www.oasis-open.org/committees/xacml/ (XACML) Version 2. Standard, OASIS, February 2005.

[MBF08] T. Mouelhi, B.Baudry, and F. Fleurey. A Generic Metamodel For Security Policies Mutation, 2008.

[MFBT08a] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon. A model-based framework for security policy specification, deployment and testing, 2008.

[MFBT08b] T. Mouelhi, F. Fleurey, B.Baudry, and Y. Le Traon. Mutating DAC And MAC Security Policies : A Generic Metamodel Based Approach., 2008.

[MX07a] E. Martin and T. Xie. A Fault Model and Mutation Testing of Access Control Policies. In *Proceedings of the 16th International Conference on World Wide Web*, pages 667–676, 2007.

[MX07b] E. Martin and T. Xie. Automated Test Generation for Access Control Policies via Change-Impact Analysis. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, 2007.

[PMT08] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-Based Tests for Access Control Policies, 2008.

[TMB07] Y. Le Traon, T. Mouelhi, and B. Baudry. Testing security policies : going beyond functional testing, 2007.

[TMPB08] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry. Test-Driven Assessment of Access Control in Legacy Applications, 2008.