M2-TIW4 sécurité des systèmes d'informations
Contrôle continu final – durée 1h30

Master Technologie de l'Information, promotion 2013 – 2014

Mercredi 18 décembre 2013

Aucun document autorisé. Le barème est indicatif. Ne pas donner de réponses sur le sujet. La concision, la précision et la clarté des réponses aux questions ouvertes font partie intégrante de l'évaluation.

1 Contrôle d'accès à rôles

Exercice 1 : Principes généraux (/5)

1. Indiquer la ou les réponses correctes : dans le contrôle d'accès basé sur les rôles
   1. Les utilisateurs disposent de privilèges qui leurs sont directement attribués ;
   2. Chaque utilisateur n'est toujours associé qu'à un seul et unique rôle ;
   3. On ne peut pas utiliser en même temps la hiérarchie de rôles et l'exclusion ;
   4. Les rôles héritent des permissions de leurs parents.
2. En formalisant les modèles à rôles hiérarchisés, avec $\mathcal{U}$ les utilisateurs, $\mathcal{R}$ les rôles, $\succeq$ un ordre partiel [1] sur $\mathcal{R}$ et $\mathcal{URA} \subseteq \mathcal{U} \times \mathcal{R}$, on définit deux applications $\mathcal{R} \to 2^{\mathcal{U}}$
   – $assigned\_users(r) = \{u \in \mathcal{U} \mid (u, r) \in \mathcal{URA}\}$
   – $auth\_users(r) = \{u \in \mathcal{U} \mid r' \succeq r \wedge (u, r') \in \mathcal{URA}\}$
   Indiquer la ou les réponses correctes en les justifiant :
   1. $auth\_users(r) \subseteq assigned\_users(r)$;
   2. Si $assigned\_users(r) = \varnothing$, alors $auth\_users(r) = \varnothing$;
   3. Si r n'a pas de descendants, alors $auth\_users(r) = assigned\_users(r)$;
   4. Si $r' \succeq r$, alors $auth\_users(r') \subseteq auth\_users(r)$;
3. Avec $\mathcal{URA} = \{(u_0, r_0), (u_0, r_1), (u_2, r_2), (u_3, r_3)\}$ et $\succeq = \{(x, x) \mid x \in \mathcal{R}\} \cup \{(r_1, r_0), (r_2, r_1), (r_2, r_0)\}$, indiquer la ou les réponses correctes en les justifiant :
   1. $assigned\_users(r_0) = assigned\_users(r_1)$;
   2. $assigned\_users(r_3) = auth\_users(r_3)$;
   3. $auth\_users(r_1) = auth\_users(r_2)$;
   4. $auth\_users(r_0) = auth\_users(r_1) \cap auth\_users(r_2)$;

1. On note $a \succeq b$ pour "a hérite de b", c-à-d quand a dispose des permissions de b.

Exercice 2 : Implémentation de RBAC avec un SGBD (/9)

Pour une application de gestion, on souhaite implémenter un mécanisme de contrôle d'accès à rôles avec un SGBD relationnel. Les entités du modèle sont les utilisateurs, les sessions, les rôles et les permissions. On suppose que le SGBD dispose de *triggers* et de contraintes `CHECK`.

1. Donner les déclarations SQL[2] d'un schéma de base de données pour stocker une politique RBAC sans hiérarchisation et sans exclusion. Préciser les clefs primaires et étrangères.
2. Donner un moyen pour assurer la contrainte d'intégrité "les utilisateurs ne peuvent endosser dans une session que des rôles qui leur sont attribués".
3. On souhaite ajouter une relation d'exclusion mutuelle entre rôles, toujours sans hiérarchie. Donner les contraintes d'intégrité associées et les moyens de les gérer.
4. Donner une requête SQL, une expression de l'algèbre relationnelle ou du calcul relationnel permettant de déterminer si un utilisateur `%u` a le droit `%p` (on considérera ces paramètres comme des constantes dans les clauses `WHERE`).
5. Même question que précédemment mais lorsque les rôles sont hiérarchisés avec un nombre de niveaux hiérarchiques fixés (e.g., 3 niveaux dans RBAC Oracle). Préciser les modifications à apporter au schéma.
6. (optionnel) Proposer une solution pour traiter des hiérachies sans limite du nombre de niveaux.

2 Politique XACML

Exercice 3 : Combinaisons de politiques XACML (/6)

Une politique de contrôle d'accès XACML en syntaxe simplifiée est donnée à la fin du sujet. En XACML, une *PolicySet* (ensemble de politiques) est composée d'une *Target* (`Null` est une cible toujours évaluée à *True*), d'une collection ordonnée de *Policy* (politiques élémentaires) et d'un algorithme de combinaison de décision (*p-o* pour *permit-overrides* dans l'exemple).Une *Policy* est construite similairement à partir de *Rule* (règles).

Une *Rule* est composée d'une décision (`p` pour `Permit`, `d` pour `Deny`), d'une *Target* et d'une *Condition*. Une *Target* est une formule de logique des prédicats sans quantificateurs ni négation n'utilisant que `subject(S)`, `action(A)` et `resource(R)`. Une *Condition* est une formule de logique de prédicats arbitraires supposées évaluables avec l'interprétation usuelle (e.g., `age(Y) < 18`). Si une requête ne correspond pas à une *Target*, le résultat de l'évaluation est `NotApplicable`.

1. Soit la requête `{subject(doctor), action(write), resource(medical_record), doctor(id,d), patient(id,p), medical_record(id,p)}`. Donner le résultat de l'évaluation de cette demande d'accès sur `PS_patient` en le justifiant.
2. Identifier un cas où `P_patient_record` et `P_medical_record` sont applicables et donner la requête correspondante.
3. Soit $\mathcal{D} = \{\texttt{Permit}, \texttt{Deny}, \texttt{NotApplicable}\}$ l'ensemble des décisions et $V = [v_0, v_1, \ldots, v_n] \in \mathcal{D}^\star$ une liste ordonnée finie de décisions avec $v_i \in \mathcal{D}$. Donner l'algorithme `d-o` en pseudo code en style impératif.
4. Commenter l'intérêt de la règle `RM2` dans `P_medical_record`.

2. La syntaxe peut être approximative tant qu'elle est compréhensible.

3 Vulnérabilités

Exercice 4 : Vulnérabilité du protocole *Wide Mouthed Frog* (/6)

Le protocole *Wide Mouthed Frog* se joue entre deux participants A et B en s'appuyant sur un serveur S. Le protocole est le suivant où T_A (resp. T_S) est une estampille temporelle choisie par A (resp. S) :

1. $A \rightarrow S : A, \langle T_A, B, K_{AB} \rangle_{K_{AS}}$
2. $S \rightarrow B : \langle T_S, A, K_{AB} \rangle_{K_{BS}}$

Il existe une faille à ce protocole qui permet à un attaquant de mettre à jour l'estampille T_S. Pour cela, l'attaque s'appuie sur une réutilisation du second message pour ré-initier le protocole en se faisant passer pour B auprès de S.

1. Préciser quels sont les éléments K_{AB}, K_{AS} et K_{BS} qui apparaissent dans ces messages et expliquer en français à sert ce protocole.
2. Décrire les messages de l'attaque en notant $I(X)$ l'attaquant qui se fait passer pour X.
3. Expliquer l'intérêt de l'attaque et ses limites.

Exercice 5 : Vulnérabilité logicielle *directory traversal* (/4)

On donne l'extrait de code PHP suivant qui est vulnérable à une attaque dite *directory traversal*.

```php
<?php
$template = 'red.php';
if (isset($_COOKIE['TEMPLATE']))
    $template = $_COOKIE['TEMPLATE'];
include ("/home/users/phpguru/templates/" . $template);
?>
```

1. Expliquer quelle est la vulnérabilité de ce code.
2. Étant donné l'exemple suivant où un cookie contenant `XXX` pour la clef `TEMPLATE` est transmis au serveur, proposer un exemple d'exploitation de la vulnérabilité.
3. Proposer différentes mesures de protection pour limiter ces attaques au niveau :
   – du traitement des paramètres dans le code PHP ;
   – de la configuration du serveur web dans l'OS hôte.

A Exemple XACML

```
PS_patient = <Null, <P_patient_record, P_medical_record>, p-o>
P_patient_record = <Null, <RP1, RP2, RP3>, d-o>
P_medical_record = <Null, <RM1, RM2>, d-o>

RP1 =
  < p,
  subject(patient) /\ action(read) /\ resource(patient_record),
  patient(id,X) /\ patient_record(id,Y) /\
  (X = Y \/ (age(Y) < 18 /\ guardian(X,Y))>

RP2 =
  < p,
  subject(patient) /\ action(write) /\ resource(patient_survey),
  patient(id,X) /\ patient_survey(id, X)>

RP3=
  < p,
  (subject(doctor) \/ subject(nurse)) /\ action(read) /\ resource(patient_record),
  true>

RM1 =
  < p,
  subject(doctor) /\ action(write) /\ resource(medical_record),
  doctor(id,X) /\ patient(id,Y) /\ medical_record(id, Y) /\ patient_doctor(Y,X)>

RM2 =
  < d,
  subject(doctor) /\ action(write) /\ resource(medical_record),
  doctor(id,X), patient(id,Y), medical_record(id, Y), not
  patient_doctor(Y,X)>
```