Livrable 4.2.2: Sécurisation des données collaboratives

Janvier 2015

LORIA

Ahmed BOUCHAMI, Olivier PERRIN

Contents

1 Gestion du contrôle d'accès aux ressources collaboratives dans OpenPaaS

Une fois l'utilisateur authentifié, sa requête d'accès à une ressource est transférée et évaluée par le service d'autorisation (contrôle d'accès). Le service d'autorisation que nous avons proposé utilise une base de donnée NoSQL MongoDB pour stocker et récupérer les règles et politiques de sécurité de la plateforme collaborative. Le modèle d'autorisation que nous avons proposé est basé sur le modèle ABAC (*Attribute Based Access Control*) avec une implantation d'un modèle formel basé sur le calcul d'événements (*Event-Calculus*). Nous utilisons le raisonneur logique DECReasoner[1] pour l'évaluation des patterns de contrôle d'accès formels que nous générons automatiquement à l'aide d'un programme développé en Java et exposé comme un service Web (utilisation de l'API JAX-WS). La figure 1 illustre les différents composants de notre plateforme de sécurité pour *OpenPaaS* et donne le déroulement du processus de contrôle d'accès.

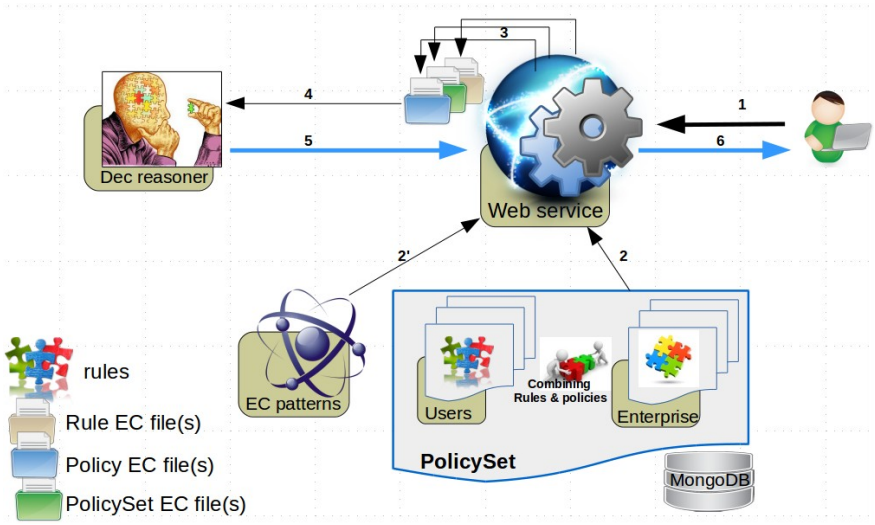

Figure 1: Architecture globale du service de contrôle d'accès d'OpenPaaS

1.1 Implantation

Comme le montre l'architecture présentée dans la figure 1, les principales étapes sont:

1. définition des règles (*rules*), politiques (*policies*) et ensemble de politiques (*policiesSets*),
2. insertion des informations (règles, politiques, ensemble de politiques) dans la base de données MongoDB,
3. récupération des informations (règles, politiques, ensemble de politiques) depuis MongoDB pour générer les patterns *Event-Calculus* correspondants,
4. envoi des patterns générés au raisonneur logique *DECReasoner* pour calculer le résultat final.

Pour chaque étape du processus, nous avons respectivement défini les classes: *AuthzStore, SaaSPatterns* et *SaaSResource*.

[1] http://decreasoner.sourceforge.net/

1.1.1 Authorization store

@Path("authzStre"): class AuthzStore
Cette partie gère les composants de contrôle d'accès, à savoir les règles, les politiques et les ensembles de politiques. Cette gestion se fait par la mise à jour (i.e insertion, modification et suppression) de la base de données MongoDB. Nous avons exposé toutes les méthodes de gestion de la base données comme un service RESTful et la mise à jour de la base est don possible via des requêtes respectant le style REST (via la soumission au service de données formatées en *JSON*[2].

Insertion

Le modèle JSON pour l'insertion d'un élément de type **rule** (figure 2).
→ *@POST, @Path("PostPath"), public Response insertRule(String msg)*

```
{
    "Subject" : userId,
    "Object" : resourceId,
    "Action" : { //the 4 actions have to be defined -> no default value
       "GET" : true,
       "POST" : true,
       "PUT" : false,
       "DELETE" : false
    },
    "RuleEffect": "Permit" || "Deny"
}
```

Figure 2: Règle au format JSON

Note: chaque action fait l'objet de création d'une règle indépendante. En d'autres termes, si nous avons par exemple : ***GET,POST=true*** *et PUT,DELETE=False,* ***deux*** *règles seront créées, une pour* ***GET*** *et l'autre pour* ***POST****.*

Le modèle JSON pour l'insertion d'un élément de type **policy** (figure 3).
→ *@POST, @Path("postPolicyPath/policyName"), Response insertPolicies(String policyAtt)*

```
{
    "PolicyName" : anID,
    "RuleName":[ ruleId1, ruleId2, ...]
}
```

Figure 3: Politique au format JSON

Réponses possibles:

- http 201 Created lorsque la règle est créée.
- http 400 Bad Request si le contenu du POST n'est pas valide.
- http 500 Server Error si une erreur imprévue est survenue pendant la création de la règle.

[2]http://www.json.org/

Mise à jour et suppression

La suppression se fait par une requête HTTP contenant l'identifiant (nom) du composant règle/politique à supprimer.
→ *@DELETE, @Path("deleteRulePath||deletePolicyPath /ruleName||policyName")* Par exemple: *http://adresseIP/SaaS/resources/authzStre/deleteRulePath/Rules1*
La mise à jour se fait également via un appel REST avec les mêmes attributs qu'une requête de type POST (figure 2 et 3).
→ *@PUT, @Path("PutPath||PutPolicyPath /ruleName||policyName")*

Les réponses possibles sont:

- http 200 OK si la règle est mise à jour.
- http 400 Bad Request si le contenu du POST n'est pas valide.
- http 404 Not Found si aucune règle n'est trouvée pour l'id 'ruleId'
- http 500 Server Error si une erreur imprévue est survenue pendant la mise à jour de la règle.

1.1.2 DECReasoner

Nous avons choisi d'utiliser un modèle formel basé sur la logique temporelle Event-calculus pour la définition de politiques, en raison de la possibilité d'analyser et de raisonner sur l'ensemble de règles, ce qui permet une vérification automatique et sûre des conflits et une expressivité plus grande et non ambiguë des scénarios.
DECReasoner[3] prend en entrée un fichier suffixé par l'extension .e et retourne tous les résultats possibles valides sous la forme de modèles (figure 4).

```
model 1:
time 0
F_ConditionSatisfied(Rule1).
F_ConditionSatisfied(Rule2).
F_ConditionSatisfied(Rule3).
F_ConditionSatisfied(Rule4).
F_RuleEffectNOTpermitted(Rule1).
F_RuleEffectNOTpermitted(Rule2).
F_RuleEffectNOTpermitted(Rule3).
F_RuleEffectNOTpermitted(Rule4).
Happens(E_DontMatchRuleParametters(Rule1), 0).
Happens(E_DontMatchRuleParametters(Rule2), 0).
Happens(E_DontMatchRuleParametters(Rule3), 0).
Happens(E_DontMatchRuleParametters(Rule4), 0).
time 1
+F_TargetDoesntHolds(Rule1).
+F_TargetDoesntHolds(Rule2).
+F_TargetDoesntHolds(Rule3).
+F_TargetDoesntHolds(Rule4).
Happens(ERuleDoesNotApply(Rule1), 1).
Happens(ERuleDoesNotApply(Rule2), 1).
Happens(ERuleDoesNotApply(Rule3), 1).
Happens(ERuleDoesNotApply(Rule4), 1).
time 2
+F_RuleNotApplicable(Rule1).
+F_RuleNotApplicable(Rule2).
+F_RuleNotApplicable(Rule3).
+F_RuleNotApplicable(Rule4).
Happens(E_PolicyDoesNotApply(Policy1), 2).
Happens(E_PolicyDoesNotApply(Policy2), 2).
time 3
+F_policyNotApplicable(Policy1).
+F_policyNotApplicable(Policy2).
Happens(E_PolicyDoesNotApply(Policy1), 3).
Happens(E_PolicyDoesNotApply(Policy2), 3).
Happens(E_policysetDontApply(PolicySet1), 3).
time 4
+F_policySetNotApplicable(PolicySet1).
```

Figure 4: Résultat du raisonneur

[3]Une documentation complète sur le raisonneur que nous avons utilisé est disponible à cet URL `http://decreasoner.sourceforge.net/csr/decreasoner.pdf`.

Le développement a consisté à implanter un intermédiaire qui crée à partir des informations stockées dans MongoDB les fichiers .e, qui les transmet au raisonneur, et qui récupère le résultat calculé par le raisonneur. Comme le raisonneur utilise Python, nous avons écrit le script qui réalise cette tâche en Python. Le nom du script est *run_decreasoner.py* (figure 5).

```
#! /usr/bin/env python

import sys
import optparse
import os
sys.path.append("/home/ahmed/decreasoner")
import decreasoner

def usage() :
    print "Usage: python %s [-o <output_file>] <input_file>" % os.path.basename(sys.argv[0])

def main() :
    open('.lock', 'w').close()

    # Option Parser
    parser = optparse.OptionParser()
    parser.add_option("-o","--output",action="store",type="string",dest="outputFilename")
    (options, args) = parser.parse_args()

    # Make sure we have our mandatory argument (file)
    if len(args) != 1 :
        print 'You must specify one file to process.'
        usage()
        os.remove('.lock')
        sys.exit(1)

    # Process
    print "The file to process is",args[0]
    if options.outputFilename :
        print "You used the --output option with value",options.outputFilename
        decreasoner.decreasoner().run(args[0],options.outputFilename)
    else :
        decreasoner.decreasoner().run(args[0],args[0]+'.res')
        outputFilename=""

    os.remove('.lock')
    sys.exit(0)

try:
    if __name__ == '__main__' :
        main()
except:
    print "Caught an exception"
```

Figure 5: Script python pour le raisonneur

Ensuite, comme notre service de contrôle d'accès est écrit en Java, nous avons implanté la méthode *decreasonerInvocation* qui établit la connexion avec le raisonneur en utilisant le script créé précédemment(*run_decreasoner.py*). La méthode *decreasonerInvocation* illustrée dans la figure 6 prend un seul paramètre*chemin*, qui est le chemin du *fichier.e* à passer au raisonneur.

Note: Dans la logique d'exécution du raisonneur DECReasonner, si une entrée commence par une minuscule, elle est interprétée comme étant une variable, alors que si elle commence par une majuscule elle est considérée comme valeur de la variable. Il est par conséquent impératif que toutes les entrées dans la base de données MongoDB respectent cette règle et commencent par une Majuscule.

1.1.3 Génération automatique des patterns Event-Calculus

Event Calculus est un langage de prédicats, et il n'est pas évident pour des utilisateurs non expérimentés d'écrire des patterns de sécurité en Event Calculus à partir des règles et politiques de sécurité récupérées

```java
import org.python.core.PySystemState;
import org.python.util.PythonInterpreter;

public void decreasonerInvocation(String chemin) {
        try {
            PythonInterpreter.initialize(System.getProperties(),
                    PySystemState.getBaseProperties(),
                    new String[]{"./lib/decreasoner/run_decreasoner.py", chemin});
            PySystemState pss = new PySystemState();
            pss.setCurrentWorkingDir("/home/ahmed/GlassFish_Server/glassfish/domains/domain1/decreasoner/");
            PythonInterpreter interp = new PythonInterpreter(null, pss);
            interp.execfile("run_decreasoner.py");
        } finally {
            System.out.println(">>> reasoning performed.");
        }
    }
```

Figure 6: Méthode Java pour l'invocation du raisonneur

souvent d'une base de données. Pour cette raison, nous avons proposé de les générer automatiquement à partir de la base de règles et politiques de sécurité (c'est-à-dire à partir de MongoDB). Pour cela, nous avons créé une méthode de génération automatique de pattern Event Calculus et nous l'avons intégrée dans notre service de contrôle d'accès. Comme le montre l'architecture globale du service de contrôle d'accès (figure 1), et plus précisément les étapes 2, 2' et 3, nous récupérons les valeurs des rules et policies (précédemment créées par les propriétaires des ressources déployées) stockées au niveau de la base MongoDB, et nous générons automatiquement les patterns adéquats. Les patterns sont des *fichier.e* et nous avons deux types de patterns:

- **les patterns génériques**: ils sont statiques et représentent l'implantation de notre modèle de contrôle d'accès. Il s'agit des patterns: **RulesPatterns, PolicyPatterns, PolicySetPatterns, sort (variable du modèle) et ordering (gestion des conflits)**. Ces patterns sont codés en dur et ne doivent pas être modifiés.
- **les patterns dynamiques**: ce sont les patterns que le service génère en fonction des informations extraites de la base de données et qu'il utilise pour chaque requête. Nous détaillons ce type de pattern dans la section suivante.

class SaaSPatterns
Les différents patterns dynamiques sont:

- **input**: contient les informations de la requête: l'utilisateur, la ressource demandée et l'action désirée.
  → *generateInputfile (String user, String resource, String action, String workingDir)*
- **rules**: contient les informations pour chaque règle dans la base de règles. Le service parcourt la base de règles et génère pour chaque règle le pattern EC associé.
  → *generateRulesPatterns(ArrayList¡String¿ rules, String rulesWorkingDir)*.
- **policy**: contient les informations pour chaque policy dans la base de policies. Le service parcourt la base de policies et génère pour chaque policy le pattern EC associé.
  → *generatePoliciesPatterns(String policies, String policiesWorkingDir)*.
- **policySet**: c'est le pattern final que le service transmet au raisonneur. Ce pattern contient un pointeur vers tous les autres patterns Event Calculus précédemment générés.
  → *generatePolicySetPatterns(StringpoliciesSet, Stringpolicies, ArrayList < String > rules, StringpoliciesSetWorkingDir, StringrulesWorkingDir, StringpoliciesWorkingDir)*.

Note: les ****WorkingDir*** *sont les emplacements où sont créés les fichier.e*

Bien que les méthodes de génération de patterns soient différentes, le principe reste toujours le même. Plus précisément, le service interroge la base de données, récupère les informations concernant les différents

composants de contrôle d'accès (rule, policy,...) sous forme de données au format *JSON*. Ensuite, le service de contrôle d'accès les analyse, met les valeurs dans des arrayList et enfin utilise ces valeurs (dynamiques) pour remplir les parties dynamiques dans patterns génériques. Nous utilisons deux méthodes pour générer les fichier.e: *writePermanentFile* et *writeTemporaryFile*. La différence réside dans le fait que la première vérifie si le fichier existe et le cas échéant n'en crée pas un nouveau et la deuxième (writeTemporaryFile) ne fait pas cette vérification. La raison pour laquelle nous avons créé la méthode *writeTemporaryFile* est que cette méthode est utilisée pour la génération du fichier final, à savoir *PolicySet.e* qui se remplit au fur et à mesure du parcours de toutes les rules et policies.
Une fois que le service de contrôle d'accès a créé le fichier *PolicySet.e* il le transmet au raisonneur via la méthode *decreasonerInvocation >> run_decreasoner.py*. Un fichier *PolicySet.e.res* est généré en conséquence. Ce dernier contient le résultat du raisonneur *DECReasoner*. Le service de contrôle d'accès lit alors le contenu du fichier *PolicySet.e.res* et récupère le résultat du raisonnement sur la requête par rapport aux règles de contrôle d'accès existantes dans la base MongoDB, et il retourne le résultat final sous forme de **boolean**. Au final, le service supprime tous les fichiers temporaires qui ont été créés pour éviter qu'ils ne soient utilisés pour une nouvelle requête (dans le cas d'une mise à jour de la base de règles ou politiques).
La classe principale qui se charge de la gestion de tout le processus de contrôle d'accès est ***class SaaSResource >> authorizations (...)***. Elle est accessible via ***@Path("authz")***. Par exemple, une requête de la part de l'utilisteur "*Member*" pour l'action *"GET"* sur la ressource *"Com1"* ressemble à:
`http://localhost:8080/SaaS/resources/authz?subject=Member&resource=Com1&action=GET`

1.1.4 Dépendances

- jython-standalone-2.5.2
- mongo-java-driver-2.12.3

1.2 Exemple

Dans l'exemple suivant, nous allons montrer comment à partir d'une règle et d'une politique définies dans une base de donnée MongoDB, nous générons les patterns event-calculus, et obtenons les résultats correspondants grâce au raisonneur DECReasoner. Nous supposons que le sujet *James* envoie une requête de demande d'accès de type *GET* sur la ressource *community1*.

Dans la figure 7, nous illustrons l'exemple d'une règle et d'une politique.

```
Rule:
{
    "_id" : ObjectId("54635d2accf2f428ef0339ae"),
    "RuleName" : "RuleMemberCom1GetEnterprise",
    "Subject" : "James",
    "Object" : "Community1",
    "Action" : "GET",
    "RuleEffect" : "Permit"
}


Policy:
{
    "_id" : ObjectId("54635dd4ccf2f428ef0339b0"),
    "PolicyName" : "PolicyMemberCom1Get",
    "RuleName" : "RuleMemberCom1GetEnterprise"
}
```

Figure 7: Exemple d'une règle et d'une politique dans une base de données MongoDB.

La figure 8 montre les patterns correspondants ainsi que le résultat calculé par le raisonneur DECReasoner sur les patterns générés automatiquement grâce à notre service web.

```
//*** Rule pattern ***
rule RuleMemberCom1GetEnterprise

[time,subject,object,action] Happens(E_MatchRuleParametters(RuleMemberCom1GetEnterprise),time) ->
    subject=James & object=Community1 & action=Get.
[time,subject,object,action] Happens(E_DontMatchRuleParametters(RuleMemberCom1GetEnterprise),time) ->
    subject!=James | object!=Community1 | action!=Get.

HoldsAt(F_RuleEffectPermitted(RuleMemberCom1GetEnterprise),0).
!HoldsAt(F_RuleEffectNOTpermitted(RuleMemberCom1GetEnterprise),0).


//*** Policy pattern***
policy PolicyMemberCom1Get
PolicyHasRules(PolicyMemberCom1Get,RuleMemberCom1GetEnterprise).


//*** Reasoning result ***
---
0
F_ConditionSatisfied(RuleMemberCom1GetEnterprise).
F_RuleEffectpermitted(RuleMemberCom1GetEnterprise).
Happens(E_MatchRuleParametters(RuleMemberCom1GetEnterprise), 0).
1
+F_TargetHolds(RuleMemberCom1GetEnterprise).
Happens(ERulePermitted(RuleMemberCom1GetEnterprise), 1).
2
+F_RulePermitted(RuleMemberCom1GetEnterprise).
Happens(E_PolicyPermitted(PolicyMemberCom1Get), 2).
3
+F_policyPermitted(PolicyMemberCom1Get).
Happens(E_policysetPermitted(PolicySet1), 3).
4
+F_policySetPermitted(PolicySet1).
---
```

Figure 8: Patterns générés à partir d'une règle et d'une politique issues d'une base de données MongoDB.