

Secteur Tertiaire Informatique
Filière « Etude et développement »

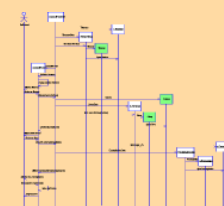
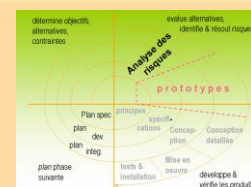
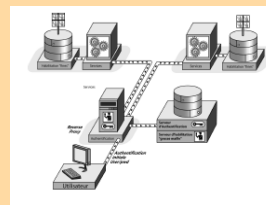
Séquence « Développer des composants métier »

Sécuriser les objets métier dans un environnement serveur

Apprentissage

Mise en situation

Evaluation



Version	Date	Auteur(s)	Action(s)
1.0	10/10/16	Lécu Régis	Création du document

TABLE DES MATIERES

Table des matières	2
1. Introduction	6
2. Résumé des principes du développement sécurisé	7
2.1 Minimiser le périmètre de l'application et privilégier la simplicité	7
2.1.1 Se limiter aux fonctionnalités indispensables	8
2.1.2 Développer des fonctionnalités simples et ciblées	8
2.1.3 Limiter le couplage.....	8
2.1.4 Utiliser l'encapsulation	8
2.2 Adopter une posture de méfiance	8
2.2.1 Considérer toute donnée externe comme potentiellement toxique.....	8
2.2.2 Sécuriser systématiquement les entrées/sorties	8
2.3 Appliquer le principe de défense en profondeur (<i>defense in depth</i>)	9
2.4 Séparer et minimiser les droits	9
2.5 Journaliser	9
2.6 Utiliser des mécanismes de sécurité existants	9
2.7 Suivre des guides de développement sécurisé	9
3. Gérer l'authentification et les droits sur les objets métiers	10
3.1 Création des utilisateurs et des groupes dans Glassfish	10
3.2 Création des rôles de sécurité.....	12
3.3 Annotation de sécurité dans les ejb.....	12
4. L'analyse statique de code	14
4.1 Introduction à l'analyse statique	14
4.1.1 Les pratiques courantes.....	14
4.1.2 Principe de l'analyse statique	14
4.1.3 Rôle de l'analyse statique	15
4.2 Les limites de l'analyse statique	15
4.2.1 Le problème de l'arrêt.....	15
4.2.2 Que peut-on faire en pratique ?	16
4.3 L'analyse statique de flux	17
4.3.1 Présentation	17
4.3.2 Le flux licite.....	18
4.3.3 Le flux illicite	18
4.3.4 Construction d'une analyse de flux	19
Sécuriser les objets métier dans un environnement serveur	

4.3.5	Exemple d'analyse de flux en Java	19
4.4	Utilisation d'un analyseur statique sous NetBeans	20
4.4.1	Installation et prise en main	20
4.4.2	Analyse d'un programme réalisé précédemment	21

Objectifs

A l'issue de cette séance, le stagiaire sera capable de :

- En partant de la conception d'une application N Tiers sécurisée, développer une couche métier sécurisée, à la fois au niveau des traitements internes et de l'accès :
 - coder les objets métier dans un style défensif
 - authentifier les utilisateurs et gérer leurs autorisations sur les objets métier
- Evaluer le niveau de sécurité requis par un objet métier, pour décider si une mise au point classique (*debug*) suffit, ou si le code du composant doit être soumis à une analyse statique.
- Connaître les atouts et les limites de l'analyse statique de code
- Pratiquer l'analyse statique d'un composant avec des outils.

Pré requis

Cette séance suppose connu le développement objet en Java, dans une architecture N Tiers (en particulier les EJB).

Méthodologie

Ce document peut être utilisé en présentiel ou à distance.

Il précise la situation professionnelle visée par la séance, la situe dans la formation, et guide le stagiaire dans son apprentissage et ses recherches complémentaires.

Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

Ressources

Sur la sécurité en Java :

- Oracle Java EE Tutorial, Part X. Security :
<https://docs.oracle.com/javaee/7/tutorial/partsecurity.htm>
- Tutorial Java de Jm Doudoux : chap. 29 sur la sécurité

Sur l'analyse statique de code :

https://www.owasp.org/index.php/Static_Code_Analysis#Description
https://fr.wikipedia.org/wiki/Analyse_statique_de_programmes

1. INTRODUCTION

Cette séance fait partie du projet **CyberEdu**, qui prend en compte la sécurité dans tout le cycle de vie du logiciel, et dans toutes les couches des applications.

Elle précise les bonnes pratiques de développement sécurisé¹, pour des objets métier déployés sur un serveur, dans une application N Tiers.

La couche métier joue un rôle central dans la stratégie de sécurité d'une application N Tiers : c'est le dernier retranchement, lorsqu'une attaque a réussi à compromettre l'interface utilisateur, en mettant ses tests en échec.

Pour être efficace, il faut sécuriser à la fois le fonctionnement interne des objets métier et leur communication avec le client :

- une validation de tous les paramètres en entrée peut éviter certaines attaques classiques de type *buffer overflow* et leur exploitation ;
- il faut authentifier les utilisateurs et leur attribuer des permissions sur les objets métier ;
- mais une bonne gestion des authentifications peut toujours être mise en échec par une usurpation d'identité (attaque de type *Man in the middle*). Il faut donc établir une liaison sécurisée entre le client et les objets métier².

L'importance de la couche métier peut justifier que ses objets soient soumis à une **analyse de code** :

- l'analyse de code est une pratique délicate, qui demande souvent une spécialisation dans la sécurité logicielle, et se situe à la limite du métier de développeur ;
- mais comme dans plusieurs autres domaines de la sécurité informatique, le développeur ne peut pas ignorer cette discipline, en la déléguant au seul spécialiste. Il doit au moins connaître ses possibilités et ses limites, les cas d'utilisation, effectuer lui-même des analyses simples ou décider de recourir à un spécialiste.

La pratique de l'analyse de code exige des connaissances théoriques :

- pour situer les limites des outils d'analyse et ne pas les utiliser de façon magique (en croyant qu'ils vont tout détecter, ou que les alarmes de sécurité sont toujours vraies et correspondent à des bugs réels) ;
- pour comprendre la complémentarité entre la mise au point (*debug*) et les différentes méthodes d'analyse de code.

Tout code, dans n'importe quelle couche, peut en théorie être soumis à une analyse mais les objets métier sont les plus appropriés car :

- ils jouent un rôle central dans le savoir-faire de l'entreprise et dans la stratégie de sécurité de l'application ;
- ils ont des entrées-sorties bien définies (boîtes noires) et n'interagissent pas directement avec l'utilisateur, ce qui facilite l'analyse de code.

¹ Vus dans la séance : « *Coder de façon défensive en suivant les bonnes pratiques de sécurité* »

² Ce dernier point est plus général et sera traité dans la séance : « *Sécuriser les couches d'une application N Tiers* »
Sécuriser les objets métier dans un environnement serveur

Cette séance doit être contextualisée selon l'environnement technique car :

- la programmation défensive repose sur des principes généraux, mais elle dépend aussi des technologies (danger du *buffer overflow* en C mais pas en Java ou en C#) ;
- l'authentification des utilisateurs, l'attribution des permissions sur les objets métier et la sécurisation des communications entre le client et le serveur sont complètement dépendantes des technologies utilisées.

Dans l'environnement Java JEE :

- les objets métier seront codés par des *ejb* ;
- on utilisera des annotations Java pour restreindre l'utilisation des *ejb* à certains utilisateurs ou groupes.

En entreprise, le développeur travaille en général seul sur ses objets métier, dont il doit fournir les tests unitaires. En s'appuyant sur le dossier de conception sécurisée de l'application, il code les objets métier, en appliquant les règles générales du développement sécurisé et en utilisant éventuellement des bibliothèques spécifiques pour sécuriser l'accès à la couche métier.

Dans ce cadre de travail, le concepteur-développeur peut être concerné par l'analyse de code, même s'il s'agit d'une pratique complexe et en évolution rapide, qui est souvent confiée à des spécialistes :

- l'équipe de conception doit identifier les objets métier qui par leur complexité, requièrent une analyse de code. Celle-ci complète les tests unitaires classiques. Ces composants doivent être prévus dès la conception pour pouvoir supporter une analyse de code ;
- dans un petit ou moyen projet, ou en PME, le développeur peut être amené à pratiquer lui-même cette analyse.

2. RESUME DES PRINCIPES DU DEVELOPPEMENT SECURISE

Nous allons à nouveau parcourir les principes du développement sécurisé, en montrant comment ils s'appliquent aux objets métier³.

2.1 MINIMISER LE PERIMETRE DE L'APPLICATION ET PRIVILEGIER LA SIMPLICITE

« *Keep Code small and simple* »

S'il est applicable à toutes les couches d'une application, y compris son interface utilisateur et ses couches techniques, ce principe s'applique en premier lieu aux objets métier, qui représentent le savoir-faire de l'entreprise.

Plus le code restera petit et simple, plus il sera ensuite facile de vérifier à la fois la cohérence fonctionnelle et la sécurité de l'application.

³ Voir le détail dans la séance : « *Coder de façon défensive en suivant les bonnes pratiques de sécurité* »

2.1.1 Se limiter aux fonctionnalités indispensables

Les fonctionnalités superflues ne font qu'augmenter la surface d'attaque : c'est d'autant plus grave pour des objets métier, publiés sur un serveur et accessibles depuis le réseau.

2.1.2 Développer des fonctionnalités simples et ciblées

Chaque classe ou méthode n'a qu'un seul objectif limité, qu'elle réalise complètement.

Cette exigence de **forte cohérence** contribue à la qualité de la couche métier, qui constitue le cœur d'une application.

Mais elle augmente aussi la résistance aux attaques, en limitant les risques d'escalade : un composant sera peut-être compromis mais l'attaque sera circonscrite à une fonctionnalité précise.

2.1.3 Limiter le couplage

Le **faible couplage** facilite l'évolutivité de l'application, en limitant les conséquences de la modification d'un composant sur les autres composants.

Mais il réduit aussi les risques d'escalade, en limitant les points d'accès et les droits du composant compromis vers les autres composants.

2.1.4 Utiliser l'encapsulation

Ce mécanisme contribue à la qualité du logiciel et à sa sécurité :

- le mécanisme d'abstraction facilite le travail du développeur, en lui permettant de manipuler des modèles simples ;
- il contribue à la défense du composant logiciel, en cachant ses mécanismes internes et leurs vulnérabilités.

2.2 ADOPTER UNE POSTURE DE MEFIANCE

2.2.1 Considérer toute donnée externe comme potentiellement toxique.

Les objets métier, publiés sur un serveur et accessibles depuis le réseau, sont particulièrement exposés :

- ils peuvent recevoir des entrées erronées ou des attaques qui auraient échappé aux filtres de l'interface utilisateur ;
- mais aussi des données falsifiées, envoyées directement par un utilisateur malveillant, qui aurait réussi à court-circuiter l'interface utilisateur normale de l'application.

2.2.2 Sécuriser systématiquement les entrées/sorties

Il faut donc sécuriser les interfaces des objets métier en validant systématiquement les types et la sémantique de toutes les entrées.

En retour, l'objet métier doit informer l'appelant du résultat du traitement par un code retour ou une exception.

2.3 APPLIQUER LE PRINCIPE DE DEFENSE EN PROFONDEUR (DEFENSE IN DEPTH)

Les objets métier constituent le cœur de l'application et son dernier retranchement (le « donjon »). Une défense périmétrique (interface utilisateur etc.) pouvant être mise en défaut, il est impératif que les objets métier ne lui fasse pas confiance et vérifient à nouveau les données en entrée.

2.4 SEPARER ET MINIMISER LES DROITS

Pour prévenir autant que possible les attaques et éviter les escalades en cas d'attaque, il faut authentifier ses utilisateurs et leur attribuer le minimum de droits sur l'application.

Un utilisateur non authentifié n'a aucun droit (interdire par défaut plutôt qu'autoriser).

Un utilisateur authentifié n'a accès qu'à ce qu'on lui autorise explicitement (utiliser des listes blanches plutôt que noires).

Puisque ce sont les objets métier qui réalisent en définitive les traitements applicatifs, il faut que l'authentification et les permissions s'appuient principalement sur eux (sans exclure des contrôles en amont dans l'interface utilisateur, et en aval dans la base de données).

2.5 JOURNALISER

Pour la même raison, comme ils effectuent les traitements les plus importants, les objets métier doivent journaliser leur action, afin de pouvoir mesurer l'étendue et les conséquences d'une attaque.

2.6 UTILISER DES MECANISMES DE SECURITE EXISTANTS

Nous allons nous appuyer sur les mécanismes de sécurité de Java EE.

2.7 SUIVRE DES GUIDES DE DEVELOPPEMENT SECURISE

Nous allons suivre le tutorial JAVA EE d'Oracle pour mettre en œuvre correctement ces mécanismes.

3. GERER L'AUTHENTIFICATION ET LES DROITS SUR LES OBJETS METIERS

Le principe « séparer et minimiser les droits » exige que chaque utilisateur soit authentifié et n'ait accès qu'aux fonctionnalités nécessaires (méthodes autorisées dans les *ejb session*).

Pour réaliser cette tâche, il va falloir :

- créer une base de données d'utilisateurs, classés par groupes ;
- associer ces utilisateurs et ces groupes à des rôles de sécurité ;
- annoter les classes et les méthodes pour restreindre l'accès de certaines méthodes à certains rôles de sécurité (et donc indirectement à certains utilisateurs ou groupes).

Nous allons nous appuyer sur le tutorial Java EE d'Oracle, partie X sur la sécurité :

<https://docs.oracle.com/javaee/7/tutorial/partsecurity.htm#GIJRP>

et en particulier le chapitre sur les applications d'entreprise (N Tiers) :

<https://docs.oracle.com/javaee/7/tutorial/security-javaee.htm#BNBYK>

3.1 CREATION DES UTILISATEURS ET DES GROUPES DANS GLASSFISH

GlassFish gère les utilisateurs avec des *realms* (royaumes, domaines).

Les principaux *realms* :

- *File* : les informations sont enregistrées dans des fichiers,
- *Certificate* : les identités sont déterminées par des certificats,
- *LDAP* : les informations sont dans un annuaire LDAP (comme *Microsoft Active Directory*),
- *JDBC* : les informations sont enregistrées dans une base de données relationnelle.

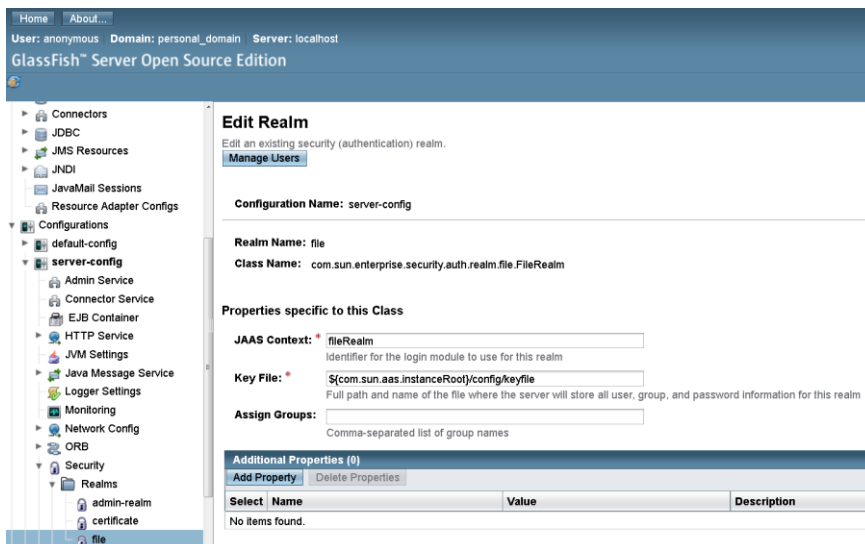
<https://docs.oracle.com/javaee/7/tutorial/security-intro005.htm#BNBXR>



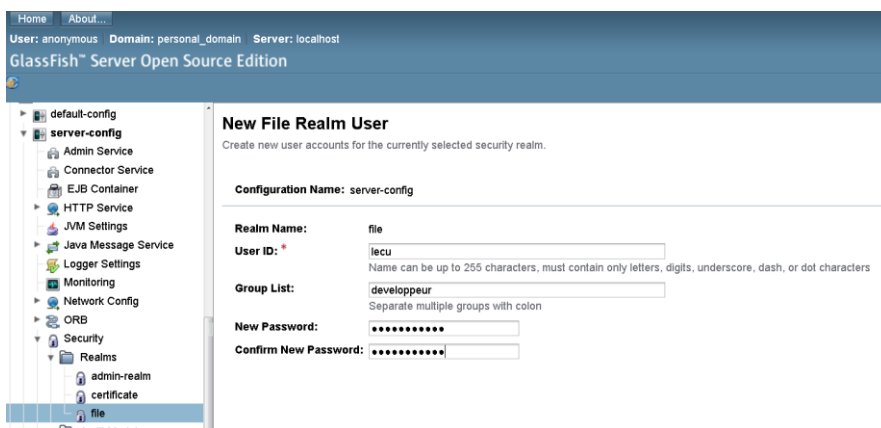
Mise en situation dans GlassFish : création d'utilisateurs et de groupe dans un realm File

Créez deux utilisateurs dans un groupe *developpeur*.

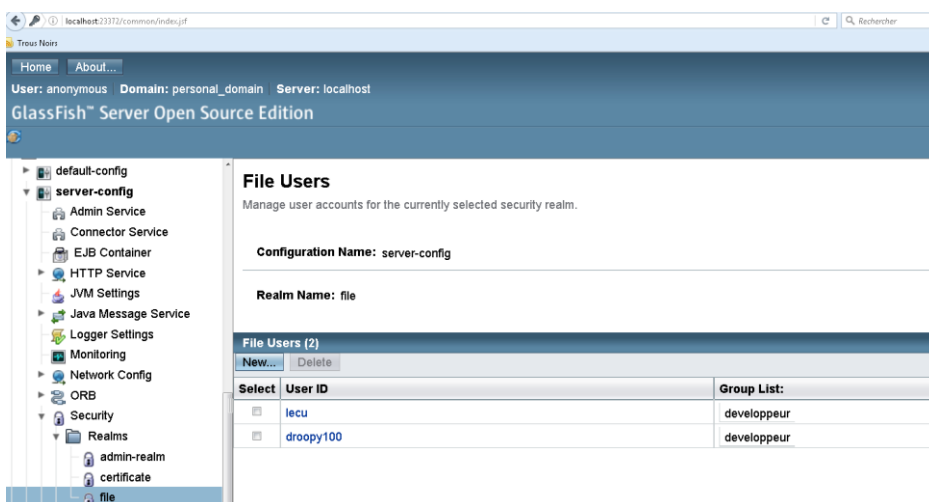
Dans la console de GlassFish, cliquez dans *Security / Realms / file*, pour ouvrir le formulaire *Edit Realm* :



Cliquez sur *Manage Users* pour ajouter des utilisateurs et des groupes, puis sur *New* :



Le nouvel utilisateur apparaît dans la liste, dans le groupe *developpeur*.



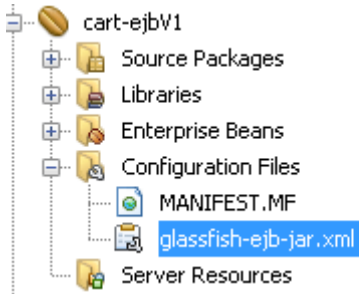
Sécuriser les objets métier dans un environnement serveur

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

3.2 CREATION DES ROLES DE SECURITE

Les permissions sur les méthodes des *ejb sessions* ne sont pas associées directement à des utilisateurs ou des groupes, mais à des rôles de sécurité qui sont associés (mappés) sur les utilisateurs et les groupes.

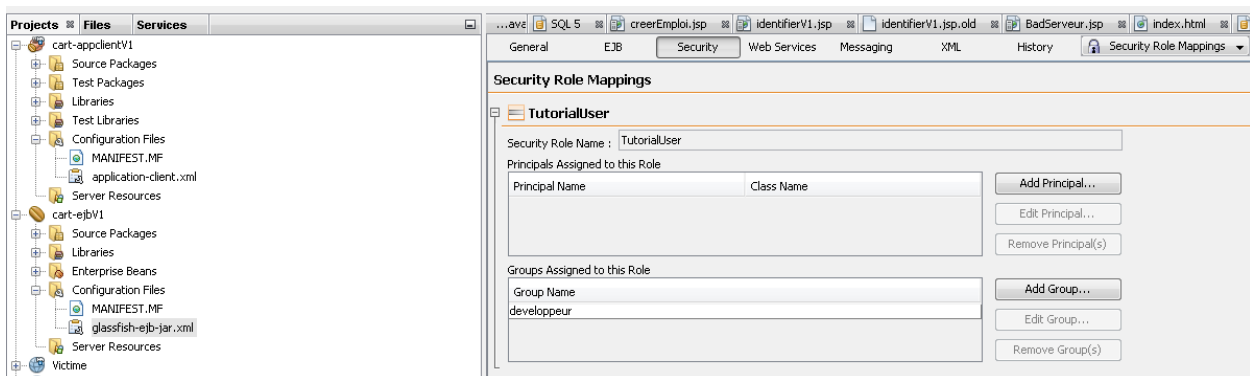
Cette association statique est définie dans le fichier de configuration *glassfish-ejb-jar.xml* :



Mise en situation dans GlassFish :

Cliquez sur le fichier *glassfish-ejb-jar.xml* et dans l'assistant, onglet *Security*, cliquez sur *SecurityRoleMappings*

Créez le rôle de sécurité *TutorialUser* et mappez-le au groupe *developpeur*



Fichier .xml obtenu :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1 EJB
3.1//EN" "http://glassfish.org/dtds/glassfish-ejb-jar_3_1-1.dtd">
<glassfish-ejb-jar>
  <security-role-mapping>
    <role-name>TutorialUser</role-name>
    <group-name>developpeur</group-name>
  </security-role-mapping>
  <enterprise-beans/>
</glassfish-ejb-jar>
```

3.3 ANNOTATION DE SECURITE DANS LES EJB

Pour utiliser un rôle de sécurité dans un *ejb*, il faut placer une annotation *@DeclareRoles* avant la déclaration de la classe :

```
@DeclareRoles("TutorialUser")
public class CartBean implements Cart, Serializable { etc.
```

Sécuriser les objets métier dans un environnement serveur

Pour restreindre l'accès à une méthode de cet *ejb* à un rôle de sécurité, il faut placer une annotation `@RolesAllowed` avant la méthode :

```
@RolesAllowed("TutorialUser")
public void addBook(String title) { etc.
```



Mise en situation dans GlassFish : version sécurisée de la maquette *Cart*

<https://docs.oracle.com/javaee/7/tutorial/security-javaee003.htm#BNBZL>

Le tutorial Java EE est contenu dans la distribution Java : Java EE SDK. Il contient des exemples qui sont commentés dans le tutorial.

Nous allons partir de la maquette *Cart* (*shopping cart*, panier sur un site de vente) qui permet d'ajouter des articles à un panier et de les lister.

Vous trouverez la version d'origine, non sécurisée dans le répertoire [sources / cart](#) :

- projet serveur : [cart-appclientV1](#)
- projet client : [cart-clientV1](#)

Ouvrez ces projets sous NetBeans et ajoutez les annotations pour restreindre les méthodes au rôle de sécurité : [TutorialUser](#), correspondant au groupe [developpeur](#), dans lequel vous avez placé vos deux utilisateurs (Corrigé dans le dossier [corriges](#)).

Au démarrage, l'application affiche un formulaire de saisie de l'utilisateur et du mot de passe :

Si la connexion réussit, l'application continue, ajoute des articles, les réaffiche, et tente de supprimer un article inexistant :

```
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
```

Dans le cas contraire, l'application s'arrête sur une exception : *iiop.login_exception*

Ce mode d'authentification est le plus simple et convient pour une démonstration.

Dans une application réelle, le stockage des utilisateurs dans un fichier est trop figé, et il faudra déclarer un *realm* lié à une base de données, par exemple Java DB.



Pour aller plus loin : la sécurisation des applications Web sous GlassFish

<https://docs.oracle.com/javaee/7/tutorial/security-webtier002.htm#GKBAA>

Sécuriser les objets métier dans un environnement serveur

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

4. L'ANALYSE STATIQUE DE CODE

Les objets métier sont le cœur d'une application : ils traduisent le savoir-faire spécifique d'une entreprise. Pour les valider au niveau fonctionnel et sécurité, il est donc utile et parfois indispensable d'aller au-delà de la mise au point à l'exécution (*debug*).

Or, un élément clé d'un processus de développement soucieux de la sécurité consiste à faire des revues de code avec l'aide d'outils. Ces outils reposent de plus en plus sur l'analyse statique de code.

4.1 INTRODUCTION A L'ANALYSE STATIQUE

4.1.1 Les pratiques courantes

a) Les tests

La pratique la plus courante en assurance logicielle repose sur les tests. On s'assure que le programme fonctionne correctement sur un jeu de test fonctionnel :

- on fournit les entrées nécessaires au fonctionnement du programme ;
- on compare les sorties réelles aux sorties attendues, pour vérifier la conformité du programme à son cahier des charges.

Les tests sont utiles, puisqu'ils peuvent révéler concrètement des erreurs qu'il faut corriger.

Mais ils sont couteux et il est difficile de couvrir tous les chemins de code.

Ils ne donnent donc aucune garantie de complétude.

Ceci est particulièrement préoccupant pour les tests de sécurité : il suffit d'un seul chemin de code non testé, d'une seule erreur pour conduire à une vulnérabilité qui peut faire tomber tout le système.

b) La revue de code

Une approche complémentaire consiste à faire auditer le code : il s'agit de convaincre un expert indépendant que le code est correct.

L'être humain est plus efficace que les tests sur machine : il peut généraliser à partir d'un seul test réel, et envisager des situations alternatives, en faisant tourner le programme « dans sa tête ».

Mais le temps humain coute beaucoup plus cher que le temps machine.

A nouveau, il s'agit d'une tâche difficile qui ne donne aucune garantie de couverture de code.

4.1.2 Principe de l'analyse statique

Un adversaire malveillant pourra tenter d'exploiter les failles que nous n'avons pas trouvées, par les tests ou la revue de code manuelle.

Que pouvons-nous faire de plus, si nous sommes préoccupés par la sécurité de notre application ?

L'analyse statique vient à notre secours.

Sécuriser les objets métier dans un environnement serveur

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

L'idée est d'utiliser un programme pour analyser le code source, sans l'exécuter.

Donc en un sens, nous demandons à l'ordinateur de faire ce que l'homme devrait faire pendant une revue de code.

Nous y gagnons en couverture de code :

- le programme d'analyse statique peut analyser de nombreuses exécutions possibles de notre programme, et même dans certains cas toutes les exécutions possibles, et garantit alors qu'une propriété de sécurité importante que nous voulons établir, est tout le temps vraie ;
- nous pouvons aussi analyser des programmes incomplets comme des bibliothèques, qui seraient sinon difficiles à tester.

Mais l'analyse statique a aussi ses inconvénients :

- elle ne peut analyser que des propriétés limitées et non pas la cohérence fonctionnelle de toute l'application ;
- elle peut omettre certaines erreurs, ou émettre de fausses alarmes ;
- elle prend beaucoup de temps de calcul.

4.1.3 Rôle de l'analyse statique

L'analyse statique joue un rôle important dans un processus de développement sécurisé, car :

- elle peut vérifier minutieusement des propriétés limitées mais utiles ;
- et donc éliminer des catégories entières d'erreurs, en permettant aux développeurs de se concentrer sur des analyses plus poussées.

L'utilisation d'un outil d'analyse statique favorise l'adoption de meilleures pratiques de développement :

- les développeurs qui utilisent un outil suivent des modèles de programmation qui évitent les erreurs évidentes ;
- cela encourage les développeurs à expliciter ce que leur programme doit faire (par des assertions et des annotations qui vont faciliter les vérifications de l'analyseur statique).

4.2 LES LIMITES DE L'ANALYSE STATIQUE

4.2.1 Le problème de l'arrêt

Un problème classique de l'analyse statique est le problème de l'arrêt (*Halting problem*) : pouvons écrire un analyseur qui puisse prouver, pour n'importe programme P et n'importe quelles entrées, que P va se terminer ?

Ce n'est malheureusement pas possible. Il a été démontré que le problème de l'arrêt n'est pas décidable : il est impossible d'écrire un analyseur général qui puisse répondre au problème de l'arrêt pour tous les programmes et toutes les entrées.



Pour aller plus loin : fr.wikipedia.org/wiki/Problème_de_l'arrêt

Si nous ne pouvons pas résoudre dans le cas général le problème de l'arrêt, qu'en est-il des autres propriétés ?

C'est peut-être faisable pour certaines propriétés de sécurité ?

Par exemple, pour un tableau `tab`, peut-on démontrer par une analyse statique que tous les accès au tableau, `tab[i]` seront dans les bornes ? Nous pourrions ainsi éliminer une cause importante de violation mémoire, les débordements de tableau.

Malheureusement, cette question est également indécidable, comme pratiquement pour toutes les propriétés intéressantes :

- une requête SQL est-elle construite à partir de données non sûres ?
- est-ce qu'un pointeur a été utilisé après la libération de la zone mémoire pointée ?
- est-ce que l'accès à une variable peut-être la cause d'une perte de données ?

4.2.2 Que peut-on faire en pratique ?

Il ne faut pas en conclure que l'analyse statique est impossible, mais qu'une analyse statique **parfaite** est impossible.

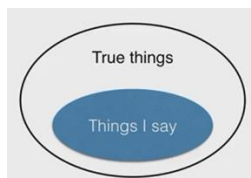
Il est tout à fait possible de réaliser une analyse statique utile, malgré :

- **la non terminaison** : dans certains cas, l'analyseur peut ne jamais se terminer ;
- **les fausses alarmes** : l'analyseur détecte des erreurs qui n'en sont pas ;
- **les omissions** : l'analyseur ne détecte pas certaines erreurs.

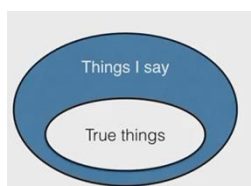
En pratique, comme les analyses sans fin perturbent l'utilisateur, les outils se limitent aux fausses alarmes et aux omissions.

Ils se situent quelque part entre l'exactitude (*soundness*) et la complétude (*completeness*) :

- **exactitude, solidité** : si l'analyse dit qu'une propriété X est vraie, alors X est vraie ;



- **complétude** : si la propriété X est vraie, alors l'analyse dit qu'elle est vraie.



Dans une analyse exacte et complète, ce que l'analyse dit recouvrerait exactement l'ensemble des choses vraies :



Sécuriser les objets métier dans un environnement serveur

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Mais nous avons vu qu'à cause de problèmes comme celui de l'arrêt, cette analyse exacte et complète ne peut exister. Il faut donc renoncer soit à l'exactitude soit à la complétude et choisir entre deux voies :

- **exactitude** : si un programme est déclaré sans erreur par l'analyseur, il l'est vraiment. Mais une alarme n'implique pas une erreur réelle (il peut y avoir des fausses alarmes)
- **complétude** : si un programme est déclaré erroné, il l'est vraiment. Mais le silence n'implique pas l'absence d'erreurs (il peut y avoir des omissions)

En réalité, les analyses les plus intéressantes ne sont ni exactes ni complètes (ni les deux), mais elles tendent vers l'un des deux. Comme l'analyse statique est impossible dans le cas général, notre but est simplement de faire un outil pratique : cela tient plus de l'art que de la science.

En particulier, il y a beaucoup d'éléments qui doivent être conciliés dans une analyse. Un outil pratique doit décider quels sont les éléments les plus importants :

- **précision** : une analyse précise va tenter de modéliser au plus près le comportement du programme, de prendre la meilleure décision pour déterminer si elle a trouvé un bug, afin d'éviter les fausses alarmes ;
- **évolutivité, extensibilité** (*scalability*) : une analyse évolutive pourra analyser avec succès des grands programmes, sans nécessiter des ressources déraisonnables, en temps d'exécution et en taille mémoire.

La plupart du temps l'évolutivité nuit à la précision, et il faut faire un compromis entre les deux objectifs.

- **clarté** : une analyse compréhensible tient compte de l'être humain : par exemple, les alarmes sont compréhensibles et facilitent les actions correctives.

Une analyse qui obéit à ces trois critères sera d'autant plus facile à réduire, que le code sera « propre », simple et clair :

- si un code est difficile à comprendre pour un être humain, il est aussi difficile à comprendre pour un outil et réciproquement ;
- l'analyse va prendre plus de temps et émettre de fausses alarmes ;
- ce problème vient de patterns de codage, ambigus ou non appropriés.

Les mauvaises performances d'un analyseur statique peuvent donc avoir un effet positif, en conduisant les développeurs à nettoyer leur code pour réduire le nombre de fausses alarmes et améliorer le temps d'exécution de l'analyseur.

Le code résultant sera aussi plus facile à comprendre pour les êtres humains.

4.3 L'ANALYSE STATIQUE DE FLUX

4.3.1 Présentation

Nous allons maintenant examiner un type particulier d'analyse statique : l'analyse de flux (*Flow Analysis*).

L'analyse de flux trace la circulation des données entre les différents emplacements mémoire dans un programme.

Ce type d'analyse permet de trouver des bugs dont la cause principale est la confiance accordée indument à des données utilisateurs non validées.

Dans une *Tainted Flow Analysis*, les entrées de l'utilisateur sont considérées comme **marquées (tainted)**.

A l'inverse, de nombreuses opérations dans le programme exigent des données sûres, **non marquées (untainted)**.

Si l'analyse de flux montre que des données **tainted** pourraient être utilisées à la place des données **untainted** attendues, cela indique une source de vulnérabilité potentielle.

Des exemples qui exigent des données sûres, **untainted** :

- la chaîne source dans un *strcpy* en C (longueur <= taille du buffer tampon)
- les champs dans un formulaire Web, utilisés pour construire une requête SQL (ils ne doivent contenir aucune commande SQL).

Le problème de l'analyse est donc de montrer qu'aucune donnée **tainted** ne sera jamais utilisée lorsque l'on attendait **untainted**.

Dans le programme à analyser :

- l'annotation **untainted** indique un récepteur de confiance,
- l'annotation **tainted** indique une source non sûre,
- l'absence d'annotation veut dire que la donnée peut être soit **tainted** soit **untainted**, et l'analyse doit déterminer ce qu'il en est.

La solution à ce problème exige de trouver tous les flux de données possibles dans le programme :

- quelles sources peuvent atteindre quels récepteurs de données ?
- y-a-t-il des flux illicites, où une source **tainted** peut circuler vers un récepteur **untainted** ?

Notre objectif est de construire une analyse exacte (*sound analysis*) : pour le problème de flux, cela signifie que si l'analyse ne trouve aucun flux illicite, il n'y en a réellement aucun.

4.3.2 Le flux licite

Un récepteur **f** qui n'est pas de confiance (**tainted**) peut accepter des données **tainted** et **untainted** :

```
void f (tainted int);  
untainted int a = ...;  
f(a);
```

Donc le flux de données **untainted** vers **tainted** est licite :

untainted <= **tainted**

4.3.3 Le flux illicite

Un récepteur **g** de confiance (**untainted**) ne peut accepter que des données **untainted** :

Sécuriser les objets métier dans un environnement serveur

```
void g (untainted int);  
tainted int b = ...;  
g(b);
```

Donc le flux de données **tainted** vers **untainted** est illicite :

tainted > **untainted**

4.3.4 Construction d'une analyse de flux

Dans le programme à analyser, il faut :

- Placer toutes les annotations **tainted** et **untainted** connues
- Placer des annotations α , β etc. pour tous les cas inconnus que l'analyse doit déterminer
- Pour chaque instruction du programme, générer des contraintes, pour déterminer les solutions possibles.

Par exemple l'affectation $x = y$, génère la contrainte $Q_y \leq Q_x$ où Q_y est le qualifieur ou annotation de la variable y et Q_x qualifie la variable x.

- Résoudre les contraintes pour trouver des solutions pour α , β etc.

Une solution est une substitution des annotations **tainted** ou **untainted**, aux annotations α , β telles que toutes les contraintes soient des flux licites

- S'il n'y a pas de solution, il peut y avoir un flux illicite.

4.3.5 Exemple d'analyse de flux en Java

```
// la méthode afficher attend en paramètre une donnée de confiance  
void afficher (untainted String s) ;  
// la méthode lire retourne une donnée non sûre  
tainted String lire () ;  
// extrait du main  
String nom = lire() ;  
String x = nom ;  
afficher (x) ;
```

- 1) Plaçons les qualifieurs sur les variables dont l'état n'est pas connu :

```
// extrait du main  
 $\alpha$  String nom = lire() ;  
 $\beta$  String x = nom ;  
afficher (x) ;
```

- 2) Déterminons les contraintes :

Sécuriser les objets métier dans un environnement serveur

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Une affectation $a = b$ est licite si $Q_b \leq Q_a$: on peut copier une donnée **tainted** dans une **tainted** ($Q_b = Q_a$), une donnée **untainted** dans une **untainted** ($Q_b = Q_a$), une donnée **untainted** dans une **tainted** ($Q_b < Q_a$) mais jamais une donnée **tainted** dans une **untainted** ($Q_b > Q_a$)

Donc :

- Comme la méthode *lire* retourne une donnée **tainted** qui est rangée dans *nom*, on a la contrainte : **tainted** \leq α
- Comme on recopie la chaîne *nom* dans la chaîne *x*, on a aussi la contrainte : $\alpha \leq \beta$
- Nous passons ensuite *x* en paramètre à la méthode *afficher*, ce qui donne : $\beta \leq$ **untainted**
- La contrainte résultante est donc : **tainted** \leq $\alpha \leq \beta \leq$ **untainted**

Ce qui est illicite car **tainted** $>$ **untainted**

Il n'y a donc pas de solution pour α et β et il peut donc s'agir d'un flux illicite.

4.4 UTILISATION D'UN ANALYSEUR STATIQUE SOUS NETBEANS

L'IDE NetBeans possède un Analyseur Statique, qui permet de détecter des problèmes possibles et des incohérences dans le code.

4.4.1 Installation et prise en main

Nous vous proposons d'installer cet analyseur dans NetBeans et de faire des essais en suivant le tutoriel : <https://netbeans.org/kb/docs/java/code-inspect.html>.

Les grandes étapes :

- téléchargez le projet *library.zip* et ouvrez-le sous NetBeans :
<https://netbeans.org/projects/samples/downloads/download/Samples/Java/library.zip>
(dossier *outils*)
Ce projet contient du code de test et l'analyseur statique *jsr305-2.0.0.jar*
- téléchargez le plugin *FindBugs* sous NetBeans



En complément du tutoriel :

Vous pouvez suivre la vidéo *code-inspect.swf* (dossier *outils*)



Mise en situation : essayez les différents analyseurs en suivant le Tutoriel

Quelques remarques pour démarrer :

Avec tous les analyseurs

- Intérêt des annotations : *@Nonnull*, *@CheckForNull* etc. pour faciliter le travail de l'analyseur statique, qui repère les variables non initialisées

Nonnull field title is not initialized by new bookstore.Book

- Détection du code inutile, pour aider le développeur à simplifier son code :

Unnecessary test for null – the expression is never null

Sécuriser les objets métier dans un environnement serveur

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

The assigned value is never used

- Formats incorrects :

Invalid value type 'String' for format specifier '%d', parameter 2

Avec le plugin FindBugs

Installez *FindBugs* dans l'éditeur *NetBeans* afin d'obtenir des précisions sur la correction des bugs relevés.

4.4.2 Analyse d'un programme réalisé précédemment

Nous allons utiliser l'analyseur (option : *All Analyser*) sur le programme de démonstration des injections SQL.

Nous essaierons d'améliorer notre code pour supprimer les alarmes.

(projet NetBeans [DemoInjectionSQL](#) dans [sources](#), corrigé [DemoInjectionSQLV2](#) dans [corrigés](#), qui revoie le code pour supprimer les alarmes)

DemoInjectionSQL

Une chaîne non constante est passée à la méthode execute d'une commande SQL.

Cette méthode invoque la méthode `execute` d'une commande SQL (statement) avec une chaîne qui semble générée dynamiquement. Préférer la préparation de la commande avec un `PreparedStatement`, c'est plus efficace et bien moins vulnérable aux attaques par injection SQL (insertion de code SQL malveillant au sein d'une requête).

- L'analyseur émet une vraie alerte sur la vulnérabilité par injection SQL dans la méthode *userExist* (requête non paramétrée) et ne lève pas d'alerte sur la version corrigée (*userExistV2*)
- Alertes sur le risque de non libération des ressources *Statement* et *ResultSet*
- Utilisation d'une chaîne de connexion avec un utilisateur et un mot de passe codés en dur (pratiques pour nos essais, mais effectivement très mauvais)
- Etc.

Sécuriser les objets métier dans un environnement serveur

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

CRÉDITS

OEUVRE COLLECTIVE DE L'AFPA

Sous le pilotage de la DIIP
et du centre sectoriel Tertiaire

EQUIPE DE CONCEPTION

Chantal PERRACHON – IF Neuilly-sur-Marne
Régis Lécu – Formateur AFPA Pont de Claix