

Secteur Tertiaire Informatique
Filière « Etude et développement »

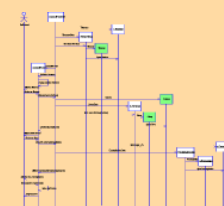
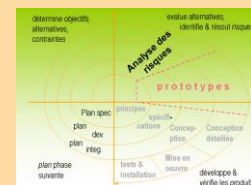
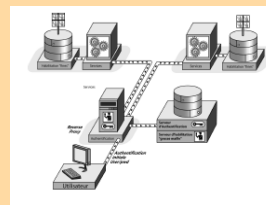
Séquence « Développer une interface utilisateur »

Coder de façon défensive en suivant les bonnes pratiques de sécurité

Apprentissage

Mise en situation

Evaluation



Version	Date	Auteur(s)	Action(s)
1.0	04/07/16	Lécu Régis	Création du document

TABLE DES MATIERES

Table des matières	2
1. Introduction	6
2. Privilégier la défense plutôt que l'attaque	6
3. Les principes du développement sécurisé.....	7
3.1 Minimiser le périmètre de l'application et privilégier la simplicité	8
3.1.1 Développer uniquement les fonctionnalités nécessaires et suffisantes	8
3.1.2 Diviser les fonctionnalités complexes en fonctionnalités courtes, simples et ciblées	8
3.1.3 Limiter autant que possible le couplage	9
3.1.4 Factoriser et réutiliser intelligemment le code	9
3.1.5 Utiliser l'encapsulation pour protéger les informations sensibles.....	10
3.2 Adopter une posture de méfiance	10
3.2.1 Considérer toute donnée externe comme potentiellement toxique.....	10
3.2.2 Sécuriser systématiquement les entrées/sorties et les interfaces	11
3.3 Appliquer le principe de défense en profondeur (<i>defense in depth</i>).....	11
3.4 Séparer et minimiser les permissions et les privilèges	11
3.4.1 Interdire par défaut plutôt qu'autoriser	12
3.4.2 Utiliser des listes blanches plutôt que noires.....	12
3.5 Journaliser	12
3.6 Maîtriser ses langages de programmation et son IDE	12
3.6.1 Connaître son langage et exploiter au mieux le compilateur et l'IDE.....	12
3.6.2 Eviter les conflits de sécurité entre le code natif et non natif	13
3.7 Renforcer la sécurité par la méthodologie et la gestion de projet	13
3.7.1 Garantir la traçabilité du code vers l'aval et vers l'amont	13
3.7.2 Un code sécurisé sera réutilisable et facile à maintenir, et inversement.....	13
3.7.3 Faire des revues de sécurité du code pendant et après le codage	13
3.8 Utiliser des mécanismes de sécurité existants et robustes.....	14
3.8.1 Eviter de réinventer la roue.....	14
3.8.2 Et utiliser correctement les mécanismes existants	14
3.9 Suivre des normes et des guides de développement sécurisé	14
4. Les règles de la programmation défensive en Java	15
4.1 Détecter l' <i>arithmetic overflow</i>	16
4.2 Interdire la désérialisation des classes non sûres.....	16
4.3 Ne pas utiliser l'introspection pour augmenter l'accessibilité	17
Coder de façon défensive en suivant les bonnes pratiques de sécurité	

4.4	Ne pas publier des attributs privés d'une classe principale à partir d'une classe imbriquée.....	18
4.5	Normaliser les chaînes en entrée avant de les valider	18
4.6	Résumé des bonnes pratiques Java qui contribuent à la sécurité	18
4.6.1	Réutiliser intelligemment le code	19
4.6.2	Valider systématiquement les entrées-sorties (types, longueur, sémantique)	19
4.6.3	Vérifier systématiquement les codes de retour et récupérer les exceptions	19
4.6.4	Evaluer les temps de réponse maximum de tous les algorithmes sensibles	20
5.	Pour aller plus loin.....	20
5.1	Règles de sécurité liées au développement objet	20
5.1.1	Limiter l'accessibilité des attributs.....	20
5.1.2	Conserver les règles de dépendance dans les classes dérivées lors d'une modification de la classe ancêtre.....	21
5.1.3	Eviter de polluer le « tas »	23
5.1.4	Fournir aux classes modifiables une fonctionnalité de copie pour pouvoir passer de façon sécurisée leurs instances à du code non sûr.....	23
5.1.5	Ne pas retourner de références vers des attributs privés de classes modifiables .	25
5.1.6	Copier de façon défensive une entrée ou un composant interne modifiable.....	25
5.1.7	Les classes avec des informations sensibles, ne doivent pas se laisser copier	26
5.1.8	Comparer les classes et pas les noms de classe.....	27
5.1.9	N'utilisez pas de champs <i>public static</i> non constants.....	27
5.2	Faites votre propre veille technologique	27
5.2.1	Sur les principes du développement sécurisé	27
5.2.2	Sur la sécurité en Java	28
6.	Conclusion	28

Objectifs

A l'issue de cette séance, le stagiaire sera capable de ¹:

- Minimiser le risque d'introduction d'une vulnérabilité dans un programme en appliquant les bonnes pratiques de développement :
 - réutilisation intelligente du code (ne pas dupliquer les fonctions)
 - vérification systématique des entrées-sorties (longueur, sémantique)
 - vérification systématique des codes de retour et récupération des exceptions
 - utilisation d'assertion (pour valider les propriétés attendues)
- Vérifier les données : type et valeur, en utilisant des listes blanches plutôt que des listes noires
- Eviter les dénis de service en vérifiant l'efficacité des tris et de tous les algorithmes sensibles (comportement en mode dégradé).

Ces objectifs seront d'abord abordés d'un point de vue général (« principes » du développement sécurisé) puis illustrés dans le contexte Java (« règles » du développement sécurisé en Java).

Pré requis

Cette séance intervient après l'apprentissage d'un langage du type Java ou C# et du développement objet. Elle suppose connues les vulnérabilités et les attaques classiques sur les langages (séance « *Identifier les spécificités de sécurité des langages et les attaques classiques* »), que l'on va maintenant chercher à éviter.

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 !

¹ Repris des objectifs de CyberEdu

Coder de façon défensive en suivant les bonnes pratiques de sécurité

Ressources

Tutorial Oracle sur le développement sécurisé en Java :

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/SecureJavaCodingGuidelines/player.html>

Top Ten » des bonnes pratiques (principes généraux, Java, C)

<https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices>

Pocket Guide (fourni) : SecureCoding_PocketGuide.pdf

developpement-securise.pdf : présentation en français sur le développement sécurisé en Java

1. INTRODUCTION

Dans la séance « *Identifier les spécificités de sécurité des langages et les attaques classiques* », nous avons vu que :

- le choix du langage est fondamental pour la sécurité d'une application : il est beaucoup plus difficile de sécuriser du C ou du JavaScript que de l'ADA ou du Java ;
- à l'inverse, aucun langage ne suffit à sécuriser une application : il faut que le développeur comprenne les vulnérabilités classiques et les combatte en permanence. Le développeur doit adopter un style particulier de programmation, la **programmation défensive** ;
- la sécurité doit être une préoccupation constante, puisqu'un seul composant logiciel défectueux (le maillon faible) peut créer une vulnérabilité globale dans l'application.

Ce style défensif ne remet pas en cause les bonnes pratiques du développement objet, vues dans les séances précédentes :

- Les principes et les règles du développement sécurisé contribuent aussi à la qualité du code et sont compatibles avec le développement objet. Inversement, en clarifiant le code, les normes et règles de développement contribuent à la sécurité.
- Mais l'approche sécurité exige d'inverser son point de vue, par rapport à l'approche fonctionnelle. On ne se contente pas de vérifier le fonctionnement standard du programme, ou sa réponse à des erreurs de l'utilisateur, mais son comportement face à des actions intentionnelles d'un utilisateur malveillant. Comment le programme réagit-il au pire :
 - entrées incohérentes entraînant des débordements de tampons ?
 - injection de code ?
 - données choisies pour ralentir les algorithmes et provoquer des dénis de service ?

C'est cette inversion de point de vue qui est visée par le projet **CyberEdu** de l'**ANSSI** que nous allons continuer à suivre pendant toute cette formation.

2. PRIVILEGIER LA DEFENSE PLUTOT QUE L'ATTAQUE

Dans cette séance, nous allons mettre en avant la défense. Il faut justifier ce choix car les attaques (« exploits ») sont plus médiatiques et peuvent être rémunératrices etc. La figure du *hacker* semble plus attirante que celle du développeur en position de défense.

Une réponse possible : nous sommes dans une formation de développement informatique et pas de *hacking* ! Mais il existe un équivalent professionnel du *hacker* en entreprise, le *pentester* qui pratique des tests d'intrusion pour évaluer la sécurité informatique.

La vraie réponse est à la fois pédagogique et professionnelle :

- la défense est fondamentalement plus difficile que l'attaque : il suffit à l'attaquant de trouver une vulnérabilité, un chemin d'attaque alors que le défenseur doit tenter de couvrir tous les chemins d'attaque ;
- l'apprentissage de la défense est plus pédagogique car elle donne une plus-value durable : les vulnérabilités finissent toujours par être corrigées alors que la programmation défensive repose sur des principes stables ;

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

- d'un point de vue professionnel, l'utilisateur de base ne se préoccupe pas spontanément de la robustesse d'un logiciel. Mais la qualité fait pourtant partie des critères qui valorisent un logiciel face à la concurrence, et certains grands éditeurs de logiciel prennent de plus en plus en compte les exigences de sécurité. Cette tendance est justifiée, car toute vulnérabilité au niveau du logiciel aura des conséquences à plus haut niveau (application, système d'information de l'entreprise). Rendre robuste un système d'information passe donc nécessairement par une action au niveau du logiciel, dès son développement².

Pour mener à bien cette défense, nous allons d'abord énoncer les principes généraux du développement sécurisé, qui concernent à la fois la phase de conception et de codage. Ces principes ont une portée générale dans le développement logiciel : ils sont applicables au développement d'interfaces graphiques, d'objets métier, d'application simple ou structurée en couches etc.

Dans un deuxième temps, nous préciserons les règles de la programmation défensive, en nous appuyant sur le langage Java : ces règles mettent en pratique les principes dans un contexte particulier. Nous montrerons en particulier comment ces règles peuvent prévenir les vulnérabilités présentées dans la séance précédente.

3. LES PRINCIPES DU DEVELOPPEMENT SECURISE

Pour présenter ces principes généraux, nous nous appuyerons sur la comparaison avec l'architecture militaire, dans les châteaux-forts médiévaux ou les forts Vauban. Les principes de défense sont les mêmes, mais ils se voient mieux dans l'architecture physique :

- Les lignes de défense : fortifications, fossés, tours etc. En logiciel, les différents composants logiciel avec leurs interfaces et la validation en entrée sur les paramètres. Les entrées incohérentes sont rejetées, comme les assaillants sont repoussés.
- Des accès limités et bien défendus (poterne, pont-levis, herse etc.) avec des contrôles sur les entrées (vérification des entrants, mots de passe etc.). En logiciel, un système d'identification des utilisateurs habilités à se servir du logiciel.
- Une architecture concentrique, avec des défenses successives indépendantes : s'ils passent la première ligne de défense (les fossés et les remparts extérieurs), les assaillants n'ont pas pour autant accès au donjon, qui constitue un château dans le château. Ce système est repris en logiciel dans le principe de « défense en profondeur » (*defense in depth*) : une entrée ne sera pas vérifiée une seule fois dans la couche externe (par exemple un formulaire web) mais dans plusieurs couches (le formulaire ET le serveur).
- Des éléments de défense les plus indépendants possibles : dans une même ligne de défense (par exemple, le rempart extérieur), chaque tour possède à nouveau son système de défense ; il ne suffit pas d'accéder au rempart et aux tours voisines pour s'en emparer. En logiciel, cela se traduit par des éléments les plus indépendants, les moins couplés possible, de façon à éviter l'escalade dans l'attaque : l'attaque réussie aura peu de suite, l'attaquant restant coincé dans le module logiciel qu'il a compromis, sans pouvoir pénétrer davantage dans le système.

² Cet argumentaire est repris de l'ANSSI, projet CyberEdu

Coder de façon défensive en suivant les bonnes pratiques de sécurité

Dans cette séance, nous ferons allusion exceptionnellement à des techniques que nous n'avons pas encore vues, pour illustrer les principes de sécurité : web, bases de données, architecture N Tiers. Dans la suite de la formation, lorsque nous pratiquerons ces techniques, nous reviendrons sur l'énoncé des principes de sécurité et sur la manière de les appliquer.

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

L'anglais est plus direct : **Keep Code small and simple.**

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

Le nombre de failles affectant des fonctionnalités avec des conséquences graves, peut être réduit de façon significative, en réduisant la taille du code qui implémente ces fonctionnalités.

Comment mettre en pratique ce principe, en réduisant et simplifiant le code ?

3.1.1 Développer uniquement les fonctionnalités nécessaires et suffisantes

Les fonctions superflues ne servent qu'aux attaquants : elles augmentent la surface d'attaque sur le logiciel et donc la probabilité qu'il soit cassé par un utilisateur malveillant.

Comment reconnaître une fonctionnalité nécessaire et suffisante ?

- Dans une gestion de projet classique (Cycle en V), ce sont celles exigées dans le cahier des charges, et détaillées dans les spécifications de votre application.
- Dans un cycle itératif et particulièrement dans les méthodes agiles, les fonctionnalités nécessaires vont être découvertes progressivement avec le client. Les méthodes agiles produisent en général des applications plus petites que le cycle en V, car le client a toujours un regard sur le logiciel en cours de développement, et il n'exige que ce dont il a réellement besoin.

3.1.2 Diviser les fonctionnalités complexes en fonctionnalités courtes, simples et ciblées

Cela rendra l'application plus facile à comprendre et à documenter, et facilitera donc la vérification de la cohérence fonctionnelle et de la sécurité de chaque composant logiciel et de la totalité de l'application.

Ce principe s'applique à haut niveau dans la définition des classes, puis dans la définition de leurs méthodes.

Le code y gagne en cohérence : chaque classe ou méthode n'a qu'un seul objectif, qu'elle réalise complètement.

Ce principe du développement sécurisé est intuitif pour un développeur objet. Il rejoint le *pattern* de conception objet de **forte cohérence** : chaque classe ou méthode fait peu de choses et le fait bien ; il vaut toujours mieux définir plusieurs petites méthodes claires et ciblées que la méthode fourre-tout qui fait n'importe quoi et très mal !

Ceci semble évident du point de vue de la conception objet, mais pourquoi est-ce aussi un principe de développement sécurisé ?

L'existence de composants logiciels volumineux, complexes et mal ciblés conduit à deux risques :

- un risque évident : les composants logiciels se défendent par leur interface, en testant leurs paramètres en entrée. Si cette défense échoue, la gravité de l'attaque dépend de la taille du composant. Imaginons un grand château-fort avec une seule ligne de défense, ou au contraire constitué de nombreuses tours avec des défenses indépendantes ;

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

- deuxième risque plus subtil : dans des composants logiciels volumineux, complexes et peu cohérents, le développeur aura une tendance regrettable à laisser dans son code des traitements inutiles ou sans rapport avec la fonctionnalité demandée, mais parfois dangereux. Ces traitements oubliés peuvent ne pas être documentés ni même testés : dans les cas extrêmes, ce sont des branches de code qui compilent mais qui n'ont jamais été exécutées. Elles sont inconnues de l'équipe de maintenance logicielle, mais tout à fait visibles pour le *hacker* qui va décompiler le code, avec une intention malveillante.

3.1.3 Limiter autant que possible le couplage

Deux composants logiciels A et B sont **couplés** si l'un (A) ne peut fonctionner sans le deuxième (B). Dans ce cas, A dépend de B et ne peut être livré et réutilisé sans B. Par exemple, pour des classes A et B : A possède au moins un attribut de la classe B ; A possède au moins une méthode avec un paramètre de la classe B ; A possède une méthode qui utilise une variable locale de la classe B.

Ce principe de développement sécurisé équivaut au *pattern* objet de **faible couplage** : construire son application en limitant le plus possible les dépendances entre les classes.

C'est une bonne pratique de conception, qui permet de supprimer, remplacer, faire évoluer certains composants logiciels, sans impacter le tout. Pourquoi est-ce aussi un principe de développement sécurisé ?

Chaque composant logiciel doit posséder une interface sécurisée, comme une tour possède un pont-levis ou une herse. L'attaquant peut compromettre cette interface sécurisée, en déjouant vos tests en entrée, comme l'attaquant physique peut casser la herse ou remplir le fossé etc. Nous sommes pour le moment au tout début de l'attaque : soit le composant logiciel (la tour) est relié à de nombreux autres composants (couplage fort) et il pourra y avoir une escalade dans l'attaque ; soit le composant est peu couplé, et l'attaquant se trouve limité dans le composant compromis. C'était le cas dans l'architecture militaire (forts Vauban) où chaque enceinte fortifiée était indépendante de la suivante.

3.1.4 Factoriser et réutiliser intelligemment le code

Keep Code small and simple. Ne jamais oublier que ce principe a un double aspect quantitatif et qualitatif.

Il faut toujours trouver un compromis entre ces deux aspects : par exemple, réduire la taille des fonctions à l'extrême va multiplier leur nombre, rendre leur interaction difficile à comprendre, et donc complexifier de façon anormale le système dans son ensemble.

La factorisation du code réduit sa taille de façon évidente, mais va-t-elle obligatoirement dans le sens de la simplicité ?

- **Factorisation intelligente** : une méthode est appelée dans plusieurs contextes, pour remplir réellement la même fonctionnalité. On ne gagne pas seulement en taille mais en clarté et en simplicité : la méthode implémente cette fonctionnalité, et rien d'autre.
- **Mauvaise factorisation** : on regroupe deux fonctionnalités A et B (indépendantes ou avec un faible recouvrement) dans un même composant logiciel C. C commence par un test sur un paramètre en entrée, qui va déterminer son comportement (fonctionnalité A ou B).

Ce type de factorisation est interdit à la fois par les bonnes pratiques de l'objet et du développement sécurisé. Même s'il réduit artificiellement la taille d'un composant, il le complexifie et le rend plus vulnérable : le *hacker* va évidemment chercher à jouer

Coder de façon défensive en suivant les bonnes pratiques de sécurité

sur le paramètre de contrôle pour exécuter la fonctionnalité B alors qu'il n'a droit qu'à A.

La réutilisation d'un code existant suit à peu près la même règle que la factorisation :

- dans un langage non objet, elle est souhaitable si le composant logiciel existant fait exactement la fonctionnalité attendue ; déconseillée, s'il faut le bricoler pour qu'il réponde aux besoins ;
- dans un langage objet, elle est plus souple : on peut par exemple dériver une classe abstraite existante A (*Paiement*) en une classe concrète B (*PaiementCheque*).

Mais il faut respecter la sémantique de l'héritage : B « est un » A, puisqu'un paiement par chèque est bien une sorte de paiement. Il faut éviter de faire une classe abstraite fourre-tout qui serve seulement à factoriser du code, sans cohérence interne.

Cette classe serait alors une cible privilégiée pour des attaques car elle serait complexe et incohérente, donc probablement vulnérable ; en tant que classe ancêtre, elle offrirait un point d'entrée dans le code des classes dérivées, une large surface d'attaque et donc une probabilité importante d'escalade dans l'attaque.

3.1.5 Utiliser l'encapsulation pour protéger les informations sensibles

L'encapsulation ne joue pas sur la taille du logiciel mais sur sa simplicité, par le mécanisme d'abstraction : l'utilisateur d'une classe n'a pas à voir et ne verra pas les détails d'implémentation de cette classe qui sont cachés (encapsulés, annotés *private*). Il ne voit que l'essentiel de la classe, ce qui permet de l'utiliser comme un outil pratique.

A nouveau, on voit que le même 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 et cohérents, sans le noyer dans les détails ;
- en sécurité, il contribue globalement à la défense du composant logiciel, puisqu'on n'est pas tenté d'attaquer ce que l'on ne voit pas !

Mais il faut préciser la portée de l'encapsulation dans la sécurité : dans la séance précédente, nous avons fourni un exemple Java montrant que ce mécanisme pouvait être détourné par l'attaquant, et qu'il était surtout utile au génie logiciel. Nous verrons dans cette séance comment utiliser le *Security Manager* de Java pour mettre en œuvre une encapsulation que l'attaquant ne puisse pas déjouer facilement.

3.2 ADOPTER UNE POSTURE DE MEFIANCE

3.2.1 Considérer toute donnée externe comme potentiellement toxique.

Un château-fort bien défendu contrôlait toutes ses entrées, au niveau du rempart extérieur : pont-levis principal, poternes, souterrains. Pour accéder au donjon, il fallait à nouveau se faire connaître, redonner un mot de passe etc.

Pour un logiciel, il faudra se préoccuper des saisies de l'interface utilisateur, mais aussi des trames réseau, de la cohérence des données dans un fichier ou une base de données externe, des informations envoyées par des capteurs ou des machines dans un environnement industriel etc.

Rappelons que beaucoup de failles graves reposent sur un laisser-aller ou une confiance excessive du développeur envers le monde extérieur :

- ne pas contrôler la taille effective d'une saisie peut provoquer un *buffer overflow* ;
Coder de façon défensive en suivant les bonnes pratiques de sécurité

- ne pas contrôler la longueur effective d'une trame réseau, ou le nombre d'octets demandé en réponse par cette trame, peut renvoyer indument des données confidentielles à l'attaquant ;
- ne pas filtrer les informations issues d'une base de données tierce, qui contient peut-être des injections SQL, peut compromettre notre propre base (=> séance « *Sécuriser l'accès et l'utilisation d'une base de données* »)

3.2.2 Sécuriser systématiquement les entrées/sorties et les interfaces

- Dans un composant logiciel, il faudra donc toujours sécuriser l'interface en considérant les entrées comme non sûres :
 - valider le type de l'entrée : si c'est une chaîne, représente-t-elle le type attendu ? si c'est un entier court, suffit-il à indexer le tableau parcouru ?
 - valider la sémantique de l'entrée : intervalles de variations, règles de gestion spécifiques.
- Inversement, dans l'appelant, il faudra toujours tester les code retour des fonctions, récupérer les exceptions etc.

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

Ce principe est une déclinaison du précédent : limiter sa confiance envers l'extérieur. Il faut :

- ne prêter aucune confiance aux données externes, qui doivent être soigneusement validées. Dans le château-fort, tout le monde était contrôlé dès la ligne de défense avancée (douve, barbacanes etc.)
- prêter une confiance limitée aux données déjà contrôlées (ou supposées l'être) par une ligne de défense externe : les personnes autorisées à pénétrer dans le donjon sont à nouveau contrôlées.

En logiciel, ce principe va permettre d'éviter l'escalade dans l'attaque : une attaque réussie n'aura qu'un effet limité sur le système global ; elle restera cantonnée dans une couche externe.

Exemple classique (=> séance « *Sécuriser l'accès et l'utilisation d'une base de données* ») : en client-serveur, une donnée doit d'abord être testée dans le client (par exemple, un formulaire web), puis à nouveau dans le serveur (par exemple, dans les paramètres en entrée d'une procédure stockée), afin d'éviter la propagation de l'attaque du client vers le serveur.

3.4 SEPARER ET MINIMISER LES PERMISSIONS ET LES PRIVILEGES

Dans le château-fort, l'architecture physique ne suffit pas à assurer une défense efficace. Celle-ci échouera si tout le monde peut faire n'importe-quoi et aller n'importe où dans le château.

Il faut des défenseurs bien identifiés, avec des privilèges et des permissions distincts et réduits au minimum selon leur fonction :

- l'identification : tous les individus sont identifiés par les gardes ou donnent un mot de passe que l'on change fréquemment ; le seigneur et ses capitaines se font reconnaître à distance, en marquant leurs messages avec un sceau ;
- privilèges séparés et minimum : chaque individu dans le château n'a le droit d'exécuter que le minimum d'actions qui correspondent à sa fonction ; par exemple, les décisions stratégiques sont réservées au seigneur ou à ses capitaines ;

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

- permissions séparées et minimales : chaque individu n'a accès qu'au minimum de ressources (provisions, armes etc.) exigées par sa fonction ; par exemple, des locaux comme la salle d'arme ne sont accessibles qu'aux soldats.

En logiciel, tout développement sécurisé exige d'identifier les utilisateurs et de gérer leurs permissions sur les ressources (par exemple les tables dans une base de données) et leurs privilèges (par exemple, créer ou supprimer une nouvelle base de données).

Comme dans le château-fort, chaque utilisateur du système ne possèdera que le minimum de privilèges et de permissions indispensables à sa fonction : cette précaution de base permettra de limiter la portée d'une attaque, en cas de compromission du compte de cet utilisateur.

3.4.1 Interdire par défaut plutôt qu'autoriser

C'est l'application de la posture de méfiance aux utilisateurs et aux ressources : la sécurité du système exige que, jusqu'à preuve du contraire (son authentification réussie), un utilisateur soit considéré comme un attaquant potentiel et n'ait donc accès à aucune ressource.

3.4.2 Utiliser des listes blanches plutôt que noires

Pour attribuer les ressources et les privilèges, il faut donc constituer explicitement des listes d'utilisateurs et de groupes d'utilisateurs autorisés (listes blanches), à l'exclusion de tous les autres.

Et éviter d'autoriser une ressource ou un privilège à tous, à l'exclusion de certains utilisateurs ou groupes d'utilisateurs réputés dangereux (listes noires).

3.5 JOURNALISER

Avec une bonne architecture physique et une bonne organisation, faut-il s'attendre à une défense parfaite ?

Il faut au contraire partir du principe qu'une attaque sera TOUJOURS possible, et **journaliser, tracer toutes les opérations**, afin de pouvoir reconstituer la séquence d'événements qui a permis l'attaque, la comprendre et améliorer son système de défense.

Il faut prévoir plusieurs niveaux de détail et de gravité dans les journaux, afin d'en faciliter l'exploitation : mémoriser tout en vrac dans l'historique, sans hiérarchiser les événements, revient à ne rien mémoriser.

3.6 MAITRISER SES LANGAGES DE PROGRAMMATION ET SON IDE

3.6.1 Connaître son langage et exploiter au mieux le compilateur et l'IDE

Dans la séance « *Identifier les spécificités de sécurité des langages et les attaques classiques* », nous avons vu que les langages sont prévus pour des objectifs différents et sont très inégaux face à la sécurité (par exemple C et Java). Pour un même langage, les compilateurs peuvent avoir un comportement différent selon les éditeurs de logiciel (le C Microsoft et le C GNU).

Un développement de qualité passe par la prise en compte de tous les défauts de programmation que peuvent détecter les compilateurs.

Il est donc nécessaire d'activer toutes les options de compilation disponibles, pour identifier les opérations dangereuses ou pour durcir le code de manière générique.

Les IDE fournissent également des compléments utiles pour la sécurité : par exemple, l'analyseur statique intégré à *Visual Studio* détecte les risques de *buffer overflow* dans le code source.

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

3.6.2 Eviter les conflits de sécurité entre le code natif et non natif

Les applications reposent souvent sur du code développé dans un autre langage : par exemple une application Java (non native) peut s'appuyer sur du code natif écrit en C, qui s'interface directement avec le matériel. C'est une vulnérabilité possible puisqu'un attaquant pourra provoquer un *buffer overflow* dans le code natif, même si la partie écrite en Java n'est pas vulnérable à ce type d'attaque.

3.7 RENFORCER LA SECURITE PAR LA METHODOLOGIE ET LA GESTION DE PROJET

On peut généraliser le principe *Keep code small and simple* : ce qui contribue à la qualité du logiciel contribue aussi à sa sécurisation.

3.7.1 Garantir la traçabilité du code vers l'aval et vers l'amont

On doit retrouver dans la conception et le code, ce qui découle de telle ou telle exigence ou expression des besoins du client. Inversement, on doit pouvoir remonter d'un extrait de code, au choix de conception et à l'exigence client qu'il implémente.

Dans une approche sécurité, cette traçabilité du code facilitera l'identification des fonctionnalités à risque, les conséquences fonctionnelles des attaques et les risques d'escalade.

Pour que cette traçabilité soit possible, il faut qu'à chaque attaque ou tentative d'attaque sur le code, corresponde une exigence dans l'analyse : un *misuse case*, cas de mauvaise utilisation, qui est le pendant du cas de bonne utilisation, le *use case* fonctionnel décrit dans la méthode UML (=> séance « Identifier les besoins de sécurité d'une application »)

3.7.2 Un code sécurisé sera réutilisable et facile à maintenir, et inversement

Il y a souvent une liaison bidirectionnelle entre la qualité et la sécurité du code. Les caractéristiques qui contribuent à la sécurité du code, simplicité, lisibilité et traçabilité, peuvent aussi contribuer à sa réutilisation et à sa maintenance. Comme le code peut être réutilisé facilement, ses propriétés de sécurité seront transférées vers d'autres projets. Si le code peut être maintenu facilement, il y aura moins de chance que des vulnérabilités soient introduites par le processus de maintenance.

Pour rentrer dans ce cercle vertueux, le développement doit toujours anticiper sur des exigences futures. Par exemple, on fera attention à ce que tous les paramètres de configuration d'une application proviennent d'une base de données ou d'un fichier de configuration, et ne soient jamais codés en dur.

Autre exemple, en Java : il vaut mieux passer des objets en paramètres, avec leurs attributs et leurs propres méthodes de vérification, que de spécifier explicitement les types des paramètres et fixer en dur les valeurs permises. Cela facilitera le développement agile, en permettant d'adapter facilement le code à de nouvelles fonctionnalités et de le réutiliser.

3.7.3 Faire des revues de sécurité du code pendant et après le codage

Le but d'une revue de sécurité du code est de découvrir et de corriger les vulnérabilités. Le rapport doit fournir assez d'informations sur les points faibles du logiciel, pour que le développeur puisse classer et établir des priorités entre les vulnérabilités, selon leur probabilité d'exploitation par des *hackers*. Quelques repères, qui seront détaillés ultérieurement dans les séances « Pratiquer une analyse de code » et « Préparer et exécuter des tests de sécurité » :

Quand faut-il placer la revue de sécurité de code ?

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

- le code devrait être vérifié au fur et à mesure qu'il est écrit, pour détecter des vulnérabilités locales dans les composants logiciels, avant leur intégration ;
- il faut aussi rechercher les vulnérabilités dans les interfaces entre les composants ;
- Il faudrait pratiquer une analyse statique du code et des tests en boîte blanche le plus tôt possible dans le cycle de développement du logiciel. Les meilleurs tests en boîte blanche sont ceux que l'on effectue sur des composants logiciels ou des blocs fonctionnels, qui peuvent être corrigés relativement facilement et rapidement, avant leur intégration. La revue de sécurité du système total portera essentiellement sur les relations entre les composants logiciels et sur leurs interfaces.

Revue par ses « pairs » : il faut toujours faire relire son code par d'autres développeurs, car un développeur a toujours beaucoup de mal à trouver ses propres erreurs.

Les outils : outils d'analyse statique et dynamique. A titre d'exemple, nous avons utilisé l'analyseur de *Visual Studio* dans la séance « *Identifier les spécificités de sécurité des langages et les attaques classiques* ».

3.8 UTILISER DES MECANISMES DE SECURITE EXISTANTS ET ROBUSTES

3.8.1 Eviter de réinventer la roue

Do not implement your own crypto.

Il ne faut pas s'improviser expert en cryptographie ou réécrire un système d'authentification maison.

On s'appuiera autant qu'il est possible sur les bibliothèques de cryptographie fournies par le langage.

On utilisera les mécanismes d'authentification du système (Windows, Unix etc.) ou du SGDB (Oracle, SqlServer etc.) ou encore un serveur d'authentification spécialisé, type RADIUS

(https://fr.wikipedia.org/wiki/Remote_Authentication_Dial-In_User_Service)

3.8.2 Et utiliser correctement les mécanismes existants

Des mécanismes éprouvés et a priori fiables peuvent être fragilisés et mis en défaut, par un mauvais paramétrage ou une utilisation impropre : il faudra donc non seulement se garder de tout amateurisme en bricolant des mécanismes de sécurité, mais acquérir une culture technique suffisante, pour bien utiliser les mécanismes existants.

3.9 SUIVRE DES NORMES ET DES GUIDES DE DEVELOPPEMENT SECURISE

Si l'on prend en compte tout ce qui précède, le développement sécurisé peut ressembler à une jungle où il est difficile de retrouver son chemin. Mais il existe heureusement des sentiers bien balisés.

Il faut choisir et utiliser un guide de développement sécurisé qui identifie explicitement les pratiques de codage sécurisé dans un langage donné. Ce guide doit décrire les failles courantes du langage et les constructions vulnérables, ainsi que leurs alternatives sécurisées.

Ces guides et normes couvrent donc à la fois ce qui devrait être fait et ce qu'il faut éviter de faire.

Exemple : le CERT fournit des guides de développement sécurisé en C, C++, Java

<https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standards+for+Java>

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

4. LES REGLES DE LA PROGRAMMATION DEFENSIVE EN JAVA

Nous allons parcourir et détailler quelques-unes des règles du CERT sur le développement sécurisé en Java.

Pour un développeur Java qui s'intéresse à la sécurité, le site du CERT sera un outil de travail utile, qu'il faut savoir déchiffrer.

Le site classe chaque règle en fonction de la gravité des conséquences, de la probabilité d'introduction d'une faille et du coût de correction d'une non observation de la règle :

Gravité (*Severity*)

Quelle est la gravité des conséquences d'un non respect de la règle ?

1 = basse (attaque par déni de service, terminaison anormale)

2 = moyenne (violation de l'intégrité des données, divulgation involontaire d'information)

3 = haute (exécution de code arbitraire, escalade de privilège)

Probabilité (*Likelihood*)

Quelle est la probabilité qu'une faille provoquée par le non respect de la règle puisse conduire à une vulnérabilité exploitable ?

1 = improbable

2 = probable

3 = forte chance

Coût de correction

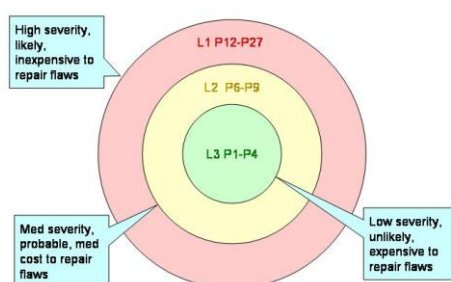
Combien cela coulera-t-il de corriger le code existant pour respecter cette règle ?

1 = élevé (détection et correction manuelles)

2 = moyen (détection automatique et correction manuelle)

3 = basse (détection et correction automatiques)

On multiplie ces trois coefficients pour chaque règle, ce qui donne un nombre entre 1 et 27 qui permet de fixer une priorité aux règles et le niveau de sécurité atteint (*level*) entre L1 et L3 : L3 signifie que l'on respecte toutes les règles (*complete compliance*) :



Pour le bon enchaînement des séances, nous allons d'abord rechercher dans les règles proposées, les solutions aux problèmes de sécurité exposés dans la séance « *Identifier les risques de sécurité des langages et les attaques classiques* ».

Nous regarderons pour chacun d'entre eux, la priorité et le niveau de sécurité de la règle correspondante.

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

4.1 DETECTER L'ARITHMETIC OVERFLOW

Rappelons que le *buffer overflow* est testé par Java et génère une exception, mais pas l'*arithmetic overflow* qui passe inaperçu.

Nous avons vu dans la séance précédente que l'*arithmetic overflow* peut rendre un programme vulnérable, en mettant en échec ses tests sur les entrées.

La règle 3 « *Numeric Types and Operations* » du CERT traite des bonnes pratiques sur les nombres en Java.

Cette règle répond au principe « *Adopter une posture de méfiance* ».

Le premier alinea *NUM00* traite de l'*integer overflow* :

- les programmes ne doivent pas permettre d'opérations mathématiques sur les entiers, qui dépassent les bornes permises par leur type.
- pour autant, Java ne lève une `ArithmeticException` que suite à une tentative de division par 0.
- c'est donc au développeur de tester la faisabilité d'une opération avant de l'effectuer, et de lever une `ArithmeticException` si elle n'est pas possible.

Le CERT propose trois correctifs possibles :

- en testant les préconditions de l'opération
- à partir de Java 8, en utilisant les fonctions sécurisées de la classe `Math`
- en faisant l'opération dans un type plus large (par exemple `long`) et en vérifiant que le résultat ne déborde pas du type initial.



En vous appuyant sur la documentation en ligne du CERT :

www.securecoding.cert.org/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow

Vous sécurisez l'exemple suivant en levant une *ArithmeticException* pour prévenir l'*overflow* et la boucle infinie qui en résulte (par les deux premières méthodes proposées par le CERT)

(fichier *DemoIntegerOverflow* dans le dossier *sources*)

```
int i = 1;
try
{
    while (i < Integer.MAX_VALUE - 1000)
    {
        i = i + 100000;
        System.out.println("i = " + i);
    }
}
catch (Exception e)
{
    System.out.println("i = " + i);
    System.out.println(e);
}
```

Proposition de solution *DemoNumericExceptionSecure* dans le dossier *corriges*

4.2 INTERDIRE LA DESERIALISATION DES CLASSES NON SURES

Nous avons vu dans la séance précédente qu'il était dangereux de désérialiser une classe inconnue (dans l'exemple, la classe *Friend*) sans être sûr qu'elle soit effectivement notre amie

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

(c'est-à-dire du code de confiance, *trusted*). Cette classe peut exécuter du code dangereux dès la désérialisation du premier objet (par exemple, dans un bloc d'initialisation statique).

La règle 14 « *Serialization* », alinéa « *Prevent deserialization of untrusted class* », définit les bonnes pratiques pour désérialiser les classes : on n'autorise la désérialisation que pour les classes de confiance, qui sont contenues dans une « liste blanche » ; pour toutes les autres classes, le programme lèvera *InvalidClassException*.

Cette règle répond au principe « *Adopter une posture de méfiance* », car une classe inconnue que l'on désérialise, appartient au monde extérieur non sûr, au même titre qu'une saisie, une trame réseau etc.



En vous appuyant sur la documentation en ligne du CERT :

<https://www.securecoding.cert.org/confluence/display/java/SER12-J.+Prevent+deserialization+of+untrusted+classes>

Vous sécurisez l'exemple fourni (dossier *sources*, fichiers *Deserial*, *Serialisation*, *Friend*) qui a été présenté dans la séance précédente.

La solution proposée par le CERT est dans l'encadré bleu « *Compliant Solution* ».

La liste blanche est stockée dans un *Set* : "GoodClass1", "GoodClass2".

Prenez soin d'indiquer le package dans le nom long de la classe : *deserialisation.Friend*

La création et l'initialisation du *Set* peuvent être simplifiées.

Proposition de solution *DemoNumericExceptionSecure* dans le dossier *corriges*.

4.3 NE PAS UTILISER L'INTROSPECTION POUR AUGMENTER L'ACCESSIBILITE

Nous avons vu dans la séance précédente qu'une utilisation abusive de l'introspection en Java peut engendrer des vulnérabilités, en rendant *public* des méthodes ou des attributs privés d'une autre classe.

La règle 15 du CERT (*Platform Security*), alinéa « *Do not use reflection to increase accessibility of classes, methods or fields* » interdit explicitement cet usage de l'introspection pour des raisons de sécurité. Cette règle dérive du principe « *Utiliser l'encapsulation pour protéger les informations sensibles* ».

Pour que cette interdiction soit testée par la machine virtuelle Java, il faut lancer le « *Security Manager* » qui contrôle les règles de sécurité standard :

```
System.setSecurityManager(new SecurityManager());
```



Consultez la documentation en ligne du CERT :

<https://www.securecoding.cert.org/confluence/display/java/SEC05-J.+Do+not+use+reflection+to+increase+accessibility+of+classes%2C+methods%2C+or+fields>

Ajoutez un « *security manager* » à l'exemple fourni (*Introspect* dans *sources*) et vérifiez que le *Security Manager* interdit d'augmenter l'accessibilité, en levant l'exception :

```
java.security.AccessControlException: access denied ("java.lang.reflect.ReflectPermission"
"suppressAccessChecks")
```

Il faut tout de même rester prudent, car une fonction publique dans une classe peut accéder à ses attributs, *public* ou *private*, par leur nom, en utilisant l'introspection. Si l'on passe ce nom en paramètre à une méthode publique, cela casse l'encapsulation.

Complétez votre maquette pour mettre en évidence ce risque : ajoutez une méthode publique `int badGet(String nom) ;`

à la classe *Secret*, qui renvoie la valeur de l'attribut dont on passe le nom

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

Proposition de solution *IntrospectSec* dans le dossier *corriges*

4.4 NE PAS PUBLIER DES ATTRIBUTS PRIVES D'UNE CLASSE PRINCIPALE A PARTIR D'UNE CLASSE IMBRIQUEE

Nous avons vu dans la séance précédente qu'une utilisation imprudente des classes imbriquées casse l'encapsulation de la classe principale.

La règle 5 du CERT (*Object Orientation*), alinéa « *Do not expose private members of an outer class from within a nested class* » interdit de publier un attribut d'une classe principale à partir d'une classe imbriquée, pour éviter de créer une vulnérabilité. Cette règle dérive du principe « *Utilisez l'encapsulation pour protéger les données sensibles* ».



Consultez la documentation en ligne du CERT :

<https://www.securecoding.cert.org/confluence/display/java/OBJ08-J.+Do+not+expose+private+members+of+an+outer+class+from+within+a+nested+class>

Changez comme préconisé par le CERT, les attributs *public* en *private*. On constate que cela n'élimine pas le problème dans le cas des classes imbriquées statiques.

Utilisez une classe imbriquée non statique, comme indiquée dans la fiche du CERT et vérifiez que dans ce cas, l'attribut *private* résout le problème (=> erreur de compilation).

Proposition de solution *InnerClassV2* et *Coordinates* dans le dossier *corriges*

4.5 NORMALISER LES CHAINES EN ENTREE AVANT DE LES VALIDER

Le principe « *adopter une posture de méfiance, sécuriser systématiquement les entrées* » s'applique en particulier aux chaînes. Mais il n'est pas si simple de les valider correctement.

Nous avons vu dans la séance précédente que l'utilisation de caractères UTF-8 dans un source Java interdisait une compréhension intuitive de ce source, et devait être proscrite.

Le problème est malheureusement plus général, puisque l'on peut aussi ranger des caractères UTF-8 dans une chaîne Java, à l'exécution : dans l'exemple du CERT, la chaîne "`\uFE64`" qui représente le caractère '<' sera affichée correctement par un *println* mais ne sera pas détectée par une comparaison directe ou une expression régulière.

Il faut donc normaliser la chaîne pour pouvoir la comparer correctement en Java

Principe de la normalisation NFKC :

https://fr.wikipedia.org/wiki/Normalisation_Unicode



Consultez la documentation en ligne du CERT :

<https://www.securecoding.cert.org/confluence/display/java/IDS01-J.+Normalize+strings+before+validating+them>

et réalisez une maquette en suivant la fiche du CERT pour mettre en évidence les problèmes posés dans la validation par les caractères UTF-8 et leur correction par la normalisation de la chaîne.

Proposition de solution *NormalizeStrings* dans le dossier *corriges*

4.6 RESUME DES BONNES PRATIQUES JAVA QUI CONTRIBUENT A LA SECURITE

Ces cas de figures méritent d'être soulignés, car ils sortent de l'ordinaire et peuvent même piéger des développeurs soucieux de la qualité de leur code.

Mais dans la majorité des cas, l'approche sécurité rejoint les bonnes pratiques de l'algorithmique et du développement objet :

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

4.6.1 Réutiliser intelligemment le code

- Ne jamais dupliquer du code (copier/coller), des méthodes ou des classes dans un projet :
 - la redondance est l'ennemi de la qualité car elle peut fausser la gestion des versions et la stratégie de test : on intègre par erreur dans un projet une vieille version d'une classe dupliquée (et oubliée !)
 - mais elle est aussi l'ennemi de l'approche sécurisée : suite à l'erreur d'intégration du code dupliqué, on réactive souvent de vieilles vulnérabilités, corrigées dans les versions ultérieures.
- Valider systématiquement le code externe :
 - tout est question de dosage : il ne faut pas réinventer la roue, mais il ne faut pas non plus intégrer dans son projet n'importe quelle bibliothèque sans garantie, ou des extraits de code sans les relire. Rappelons les bonnes pratiques :
 - pour les bibliothèques, s'en tenir aux éditeurs connus ou à des logiciels libres sérieux
 - face à un problème complexe ou nouveau, il est intéressant de trouver un exemple qui tourne. Mais cet exemple doit être testé dans un programme de test isolé du développement principal.
 - s'il répond au problème posé, il est préférable de le comprendre et de le reconstruire entièrement plutôt que de le recopier dans le projet final, car une lecture même attentive d'un code externe ne voit jamais tout (*back door*, *virus* etc.)

4.6.2 Valider systématiquement les entrées-sorties (types, longueur, sémantique)

Les exemples précédents (*Arithmetic Overflow*, Normalisation des chaînes) montrent que tout est lié, et qu'il faut construire une véritable stratégie de défense des entrées :

- Définir les paramètres de type entier avec le type entier le plus grand du langage (en java `BigInteger`) pour éviter les *overflow* dans les passages de paramètres, et tester efficacement les limites attendues. On pourra ainsi protéger réellement les données (ne pas accéder à une partie protégée d'un tableau etc.)
- Utiliser des listes blanches plutôt que des listes noires : l'exemple sur les chaînes a montré qu'il est difficile de tout prévoir (différents jeux de caractères) et qu'un *hacker* peut tenter de plusieurs façons d'introduire des séquences interdites dans une chaîne. Il vaut donc mieux travailler avec des listes blanches, contrôlées par des expressions régulières qui acceptent uniquement ce qui est PERMIS.

4.6.3 Vérifier systématiquement les codes de retour et récupérer les exceptions

Il faut prendre autant de précautions avec le traitement des sorties dans l'appelant (au sens large : paramètres, valeurs de retour, récupération d'exception, écriture dans des fichiers ou des bases de données) que sur le filtrage des entrées dans la méthode appelée :

- si une méthode détecte une incohérence sur ses entrées et retourne *false* mais que son appelant ne teste pas ce code retour, cela introduit une vulnérabilité muette ;
- pour des méthodes sensibles, on privilégiera donc les exceptions métier : si l'appelant est négligent et ne traite pas l'exception, cela créera un *bug* visible qui devra être corrigé.

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

4.6.4 Evaluer les temps de réponse maximum de tous les algorithmes sensibles

Un développeur consciencieux teste ses algorithmes sensibles (par exemple un tri) avec un jeu d'essai fonctionnel, comportant les bons cas et les cas d'erreurs classiques. Mais ce jeu d'essai est souvent réduit en taille, et prévoit rarement les cas tordus utilisés dans les attaques.

Il faut donc compléter l'approche de test fonctionnel en :

- réalisant des tests quantitatifs, avec des volumes d'information plus importants que ceux attendus ;
- recherchant dans la littérature technique le comportement de l'algorithme en mode dégradé (données volumineuses ou inattendues).
- existe-t-il des situations où l'algorithme peut ralentir l'application/le service de façon significative et provoquer un déni de service ?

5. POUR ALLER PLUS LOIN

Nous avons vu comment corriger les vulnérabilités courantes, découvertes dans la séance précédente (sauf celles qui résultaient d'erreurs de programmation pures et simples, qu'il suffit d'éviter).

Il n'est pas possible dans cette initiation de détailler toutes les règles du CERT mais nous allons parcourir la règle 5 « *Object Orientation* » qui est au centre de notre métier et rejoint les bonnes pratiques présentées dans les séances sur le développement objet :

<https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=6029486>

N'hésitez pas à lire le détail et les exemples fournis par le CERT si vous avez des doutes sur certaines de ces règles.

5.1 REGLES DE SECURITE LIEES AU DEVELOPPEMENT OBJET

5.1.1 Limiter l'accessibilité des attributs

Limit accessibility of fields :

L'encapsulation répond aux principes « *privilégier la simplicité* » et « *adopter une posture de méfiance* » :

- Simplicité (abstraction) : on ne voit que ce qui est utile à la compréhension de la classe et non ses rouages internes, ce qui est favorable au développement sécurisé, car on n'est pas tenté d'attaquer ce que l'on ne voit pas ;
- Posture de méfiance, confiance minimale, validation systématique des entrées : les attributs privés ne peuvent être modifiés que par des *set*, après un contrôle de cohérence des paramètres en entrée ;
- Rappel : il faut lancer le *Security Manager* de Java pour garantir que l'encapsulation soit effective et ne puisse pas être cassée par le mécanisme d'inspection.

(code vulnérable dans l'encadré rouge, solution dans l'encadré vert)

```
public class Widget {  
    public int total;                // Nombre d'éléments  
  
    void add() {  
        if (total < Integer.MAX_VALUE)  
        {  
            total++;  
            // ...  
        }  
    }  
}
```

Coder de façon défensive en suivant les bonnes pratiques de sécurité

```

    }
    else
    {
        throw new ArithmeticException("Overflow");
    }
}

```

```

public class Widget {
    private int total;                // déclaration en privé

    public int getTotal () {
        return total;
    }
    // add() ne change pas
}

```

5.1.2 Conserver les règles de dépendance dans les classes dérivées lors d'une modification de la classe ancêtre

Preserve dependencies in subclasses when changing superclasses

- Répond au principe « *privilégier la simplicité* » : la cohérence d'une arborescence de classes doit reposer sur des règles simples qui sont respectées par toutes les classes, et par toutes leurs instances, dans leurs différents états : des invariants de programmation.
- Lorsqu'un développeur modifie une classe ancêtre, il doit s'assurer que ces modifications conservent tous les invariants de la classe ancêtre, dont dépendent aussi les classes dérivées.
- Ne pas respecter ces invariants pourrait causer des vulnérabilités : par exemple, le développeur de la classe dérivée a l'habitude de ne pas faire certaines vérifications qui sont faites et garanties par la classe ancêtre ; il est permis de factoriser et de déléguer certaines vérifications (par exemple dans une classe abstraite) mais qu'arrive-t-il si on retire ses vérifications sans en mesurer les conséquences dans l'arborescence des classes dérivées ?

```

class Account {    // en charge de toutes les données bancaires, comme le solde
    private double balance = 100;        // solde du compte
    // retrait si le solde le permet
    boolean withdraw(double amount) {
        if ((balance - amount) >= 0)
        {
            balance -= amount;
            System.out.println("Withdrawal successful. The balance is : "
                               + balance);
            return true;
        }
        return false;
    }
}
// version d'origine de la classe dérivée
public class BankAccount extends Account {
    // la classe derive prend en charge l'authentification
    @Override boolean withdraw(double amount) {
        if (!securityCheck())
        {

```

Coder de façon défensive en suivant les bonnes pratiques de sécurité

```

        // lève une exception si tentative d'intrusion
        throw new IllegalAccessException();
    }
    // mais confie le retrait à la classe ancêtre
    return super.withdraw(amount);
}
private final boolean securityCheck() {
    // Check that account management may proceed
}
}
public class Client {
    public static void main(String[] args) {
        Account account = new BankAccount();
        // Enforce security manager check
        boolean result = account.withdraw(200.0);
        System.out.println("Withdrawal successful? " + result);
    }
}

```

```

// La deuxième version ajoute une gestion de l'autorisation de découvert
// (overdraft) dans la classe ancêtre, sans précaution particulière dans la classe dérivée
class Account
{
    // Maintains all banking-related data such as account balance
    private double balance = 100;

    // gestion du découvert
    boolean overdraft()
    {
        balance += 300;    // ajoute 300 s'il y a un découvert
        System.out.println("Added back-up amount. The balance is : "
                           + balance);
        return true;
    }
    // Other Account methods
}

public class BankAccount extends Account {
    // la classe dérivée gère l'authentification.
    // Inchangée : il manque la redéfinition de la nouvelle méthode : overdraft
}
// utilisation licite, mais qui crée une faille de sécurité
public class Client {
    public static void main(String[] args) {
        Account account = new BankAccount();
        // Enforce security manager check
        boolean result = account.withdraw(200.0);
        // si compte en découvert autorisée, on ajoute 300 par overdraft
        if (!result) {
            result = account.overdraft();
        }
        System.out.println("Withdrawal successful? " + result);
    }
}

```



```
// Client malveillant ! outrepassé l'authentification en appelant directement
// overdraft
public class MaliciousClient {
    public static void main(String[] args) {
        Account account = new BankAccount();
        // No security check performed
        boolean result = account.overdraft();
        System.out.println("Withdrawal successful? " + result);
    }
}
```

Solution sécurisée : on interdit l'appel de *overdraft* dans *BankAccount* en redéfinissant la méthode et en levant une exception :

```
class BankAccount extends Account {
    // ...
    @Override boolean overdraft() { // Override
        throw new IllegalArgumentException();
    }
}
```

5.1.3 Eviter de polluer le « tas »

[Prevent heap pollution](#)

Ce problème survient quand une variable d'un certain type référence un objet d'un autre type. Par exemple, lorsque l'on mélange des collections typées et non typées : *List* et *List<String>*

5.1.4 Fournir aux classes modifiables une fonctionnalité de copie pour pouvoir passer de façon sécurisée leurs instances à du code non sûr

[Provide mutable classes with copy functionality to safely allow passing instances to untrusted code](#)

- Le non respect de cette règle casse l'encapsulation : du code externe peut modifier l'instance ou les attributs d'une classe modifiable.
- Il faut donc créer des copies de ces classes pour pouvoir passer les instances copiées au code non sûr.
- Le CERT propose plusieurs méthodes classiques pour réaliser cette copie :
 - o par un constructeur de copie,
 - o en utilisant une méthode statique qui implémente le pattern *factory* (fabrique)
 - o en redéfinissant la méthode *clone* de la classe *Object*.

```
public final class MutableClass {
    private Date date;

    public MutableClass(Date d) {
        this.date = d;
    }

    public void setDate(Date d) {
        this.date = d;
    }
}
```

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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


```

    public Date getDate() {
        return date;
    }
}

```

```

// Solution avec constructeur de copie
public final class MutableClass
{
    private final Date date;                // la date est constante

    public MutableClass(MutableClass mc)    // constructeur de copie
    {
        this.date = new Date(mc.date.getTime());
    }
    public MutableClass(Date d)
    {
        this.date = new Date(d.getTime()); // copie défensive de la date
    }
    public Date getDate() {
        return (Date) date.clone();        // on retourne une copie (clone)
    }
}

```

```

// solution avec le pattern factory
class MutableClass
{
    private final Date date;

    private MutableClass(Date d)           // non instantiable et non héritable
    {
        this.date = new Date(d.getTime()); // faire une copie défensive
    }
    public Date getDate()
    {
        return (Date) date.clone();        // retourner une copie
    }

    // pattern factory
    public static MutableClass getInstance(MutableClass mc)
    {
        return new MutableClass(mc.getDate());
    }
}

```

```

// solution par clonage
public final class MutableClass implements Cloneable {
    private Date date;

    public MutableClass(Date d) {
        this.date = new Date(d.getTime());
    }

    public Date getDate() {
        return (Date) date.clone();
    }
}

```

Coder de façon défensive en suivant les bonnes pratiques de sécurité

```

    }

    public void setDate(Date d) {
        this.date = (Date) d.clone();
    }

    public Object clone() throws CloneNotSupportedException {
        final MutableClass cloned = (MutableClass) super.clone();
        cloned.date = (Date) date.clone(); // copier manuellement l'objet date modifiable
        return cloned;
    }
}

```

5.1.5 Ne pas retourner de références vers des attributs privés de classes modifiables

[Do not return references to private mutable class members](#)

- Le non respect de cette règle casse l'encapsulation
- Passer la référence d'un attribut privé modifiable permettrait à un appelant malintentionné de modifier indument cet attribut privé, contre l'intention du concepteur de la classe et ses choix d'abstraction et de sécurité. Par exemple, la date contenue dans *MutableClass* pourrait être modifiée indument par la référence renvoyée par la méthode *getDate*. Il faut donc renvoyer une copie de la date, grâce à la méthode *clone*

```

class MutableClass {
    private Date d;

    public MutableClass() {
        d = new Date();
    }

    public Date getDate() {
        return d;
    }
}

```

```

public Date getDate() {
    return (Date)d.clone();
}

```

5.1.6 Copier de façon défensive une entrée ou un composant interne modifiable

[Defensively copy mutable inputs and mutable internal components](#)

Cette règle sert le principe « *adopter une posture de méfiance* », en sécurisant systématiquement les entrées :

- Si une entrée est modifiable par un acteur externe non digne de confiance (*not trusted*), celui-ci peut très bien modifier l'entrée à tout moment, après que l'application sécurisée l'ait validée, ce qui fait échouer le contrôle en entrée.
- Autre cas : l'état de l'entrée modifiable change de lui-même : par exemple, dans la fiche du CERT, un *cookie* est encore valide en début de fonction, lors du test de sa date
Coder de façon défensive en suivant les bonnes pratiques de sécurité

d'expiration et dépasse cette date avant la fin de la fonction, ce qui fausse la cohérence du traitement. Remarquons que la solution du CERT repose une nouvelle fois sur le clonage : on copie le *cookie* dans une variable de travail ; on teste et on utilise la copie et non l'original modifiable, pour garantir la cohérence du traitement

- Ce type d'incohérence sur des données modifiables entre le moment du test et le moment de l'utilisation est une cause fréquente de vulnérabilités, désignée par l'abréviation *TOCTOU* : *Time of check to time of use*.

https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

Le code ci-dessous contient une vulnérabilité de type TOCTOU puisqu'un attaquant peut tenter de faire expirer le cookie entre le test de sa date et le moment de son utilisation. En encadré rouge la vulnérabilité, en vert la correction proposée par le CERT :

```
public final class MutableDemo {
    // java.net.HttpCookie is mutable
    public void useMutableInput(HttpCookie cookie) {
        if (cookie == null)
        {
            throw new NullPointerException();
        }
        // Check whether cookie has expired
        if (cookie.hasExpired())
        {
            // Cookie is no longer valid; handle condition by throwing an exception
        }
        // Cookie may have expired since time of check
        doLogic(cookie);
    }
}
```

```
public final class MutableDemo {
    // java.net.HttpCookie is mutable
    public void useMutableInput(HttpCookie cookie) {
        if (cookie == null) {
            throw new NullPointerException();
        }

        // Create copy
        cookie = (HttpCookie)cookie.clone();

        // Check whether cookie has expired
        if (cookie.hasExpired()) {
            // Cookie is no longer valid; handle condition by throwing an exception
        }

        doLogic(cookie);
    }
}
```

5.1.7 Les classes avec des informations sensibles, ne doivent pas se laisser copier

[Sensitive classes must not let themselves be copied](#)

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

C'est presque une lapalissade : les classes qui contiennent des secrets, des clés privées et toute information confidentielle doivent REFUSER d'être copiées, en redéfinissant la méthode *clone* de la classe *Object*, pour lever une exception *CloneNotSupportedException*

```
class SensitiveClass {
    // ...
    public final SensitiveClass clone() throws CloneNotSupportedException
    {
        throw new CloneNotSupportedException();
    }
}
```

5.1.8 Comparer les classes et pas les noms de classe

Compare classes and not class names

Cette règle suit le principe « *adopter une posture de méfiance* » : supposons qu'une classe de confiance A ait besoin de déléguer des traitements à une autre classe de confiance B, le minimum est de s'assurer qu'on a bien chargé B et non un homonyme non sûr.

Hors, dans une Machine Virtuelle Java, « deux classes sont réellement identiques, si elles sont chargées par le même chargeur de classe (*class loader*) et qu'elles ont le même nom pleinement qualifié (c'est-à-dire avec son package en préfixe). Deux classes avec le même nom mais des noms de package différents sont différentes, ainsi que deux classes avec le même nom pleinement qualifié mais chargées par des *class loader* différents ».

```
// vérifie si l'objet auth est de la classe attendue
if (auth.getClass().getName().equals(
    "com.application.auth.DefaultAuthenticationHandler")) {
    // ...
}
```

```
// vérifie si l'objet auth est de la classe attendue
if (auth.getClass() == com.application.auth.DefaultAuthenticationHandler.class) {
    // ...
}
```

5.1.9 N'utilisez pas de champs *public static* non constants

Do not use public static nonfinal fields

Le code appelant peut accéder directement aux champs *public static* puisque l'accès à ces champs n'est pas testé par le *Security Manager*.

Les nouvelles valeurs ne peuvent pas être validées par code avant d'être stockées dans ces champs. Dans le cas d'une application *multithread*, le partage des champs *public static* n'est pas géré et peut aboutir à des états incohérents.

5.2 FAITES VOTRE PROPRE VEILLE TECHNOLOGIQUE

5.2.1 Sur les principes du développement sécurisé

Pour ceux qui n'ont pas peur de l'anglais, le document fourni *SecureCoding_PocketGuide* liste les bonnes pratiques du développement sécurisé, et contient de nombreux liens qui renvoient aux sites de référence.

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

5.2.2 Sur la sécurité en Java

- Un document en français (fourni) : *developpement-securise* (bonne utilisation des exceptions en Java, tests unitaires avec junit, utilisation du *Security Manager* etc.)
- Le Tutorial Oracle sur le développement sécurisé en Java :
<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/SecureJavaCodingGuidelines/player.html>
Conseillée : vidéo en anglais, très claire, facile à utiliser, et très détaillée.
- En complément du CERT, on peut se perfectionner en sécurité java, grâce au site d'Oracle : <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html>
- Et en particulier grâce au guide du développement sécurisé qui développe les points vus dans cette séance : <http://www.oracle.com/technetwork/java/seccodeguide-139067.html>
- Pour l'utilisation de Java en développement Web dans les séances ultérieures, on suivra le site d'OWASP : <https://www.owasp.org/index.php/Category:Java>

6. CONCLUSION

Quelques points de repère dans notre formation au développement sécurisé.

- Cette séance fait suite à « Identifier les spécificités de sécurité des langages et les attaques classiques », en exposant les principes et les règles qui doivent être mis en œuvre pour prévenir les attaques classiques.
- Les principes du développement sécurisé, vus dans cette séance, sont généraux et seront également mis en œuvre dans les séances :
 - Sécuriser l'interface utilisateur,
 - Sécuriser l'accès et l'utilisation de la base de données,
 - Identifier les failles de sécurité et appliquer les bonnes pratiques de sécurisation des applications Web,
 - Concevoir une application N Tiers sécurisée
 - Sécuriser les composants métiers dans un environnement serveur
 - Sécuriser les couches d'une application N Tiers
- Le développement sécurisé en langage Java sera complété dans les séances :
 - Utiliser la cryptographie et les mécanismes de sécurité du Web (=> utilisation de l'API de cryptographie de Java)
 - Sécuriser les composants métiers dans un environnement serveur
 - Sécuriser les couches d'une application N Tiers

Coder de façon défensive en suivant les bonnes pratiques de sécurité

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

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconque. »