Afpa

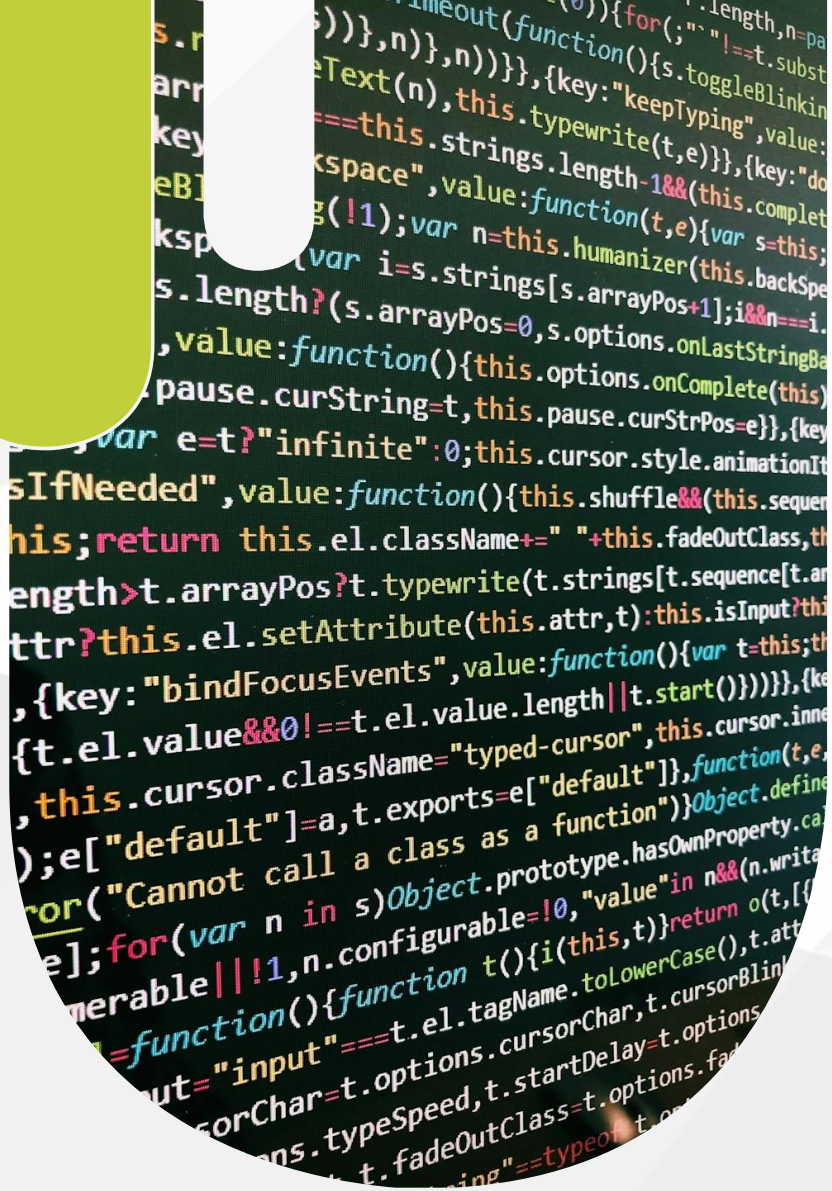
se former, avancer

Codage défensif

Apprentissage

Concepteur Développeur d'Applications

Introduction



introduction

Il est important de faire les bons choix

- 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
- A 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.

Les bonnes pratiques

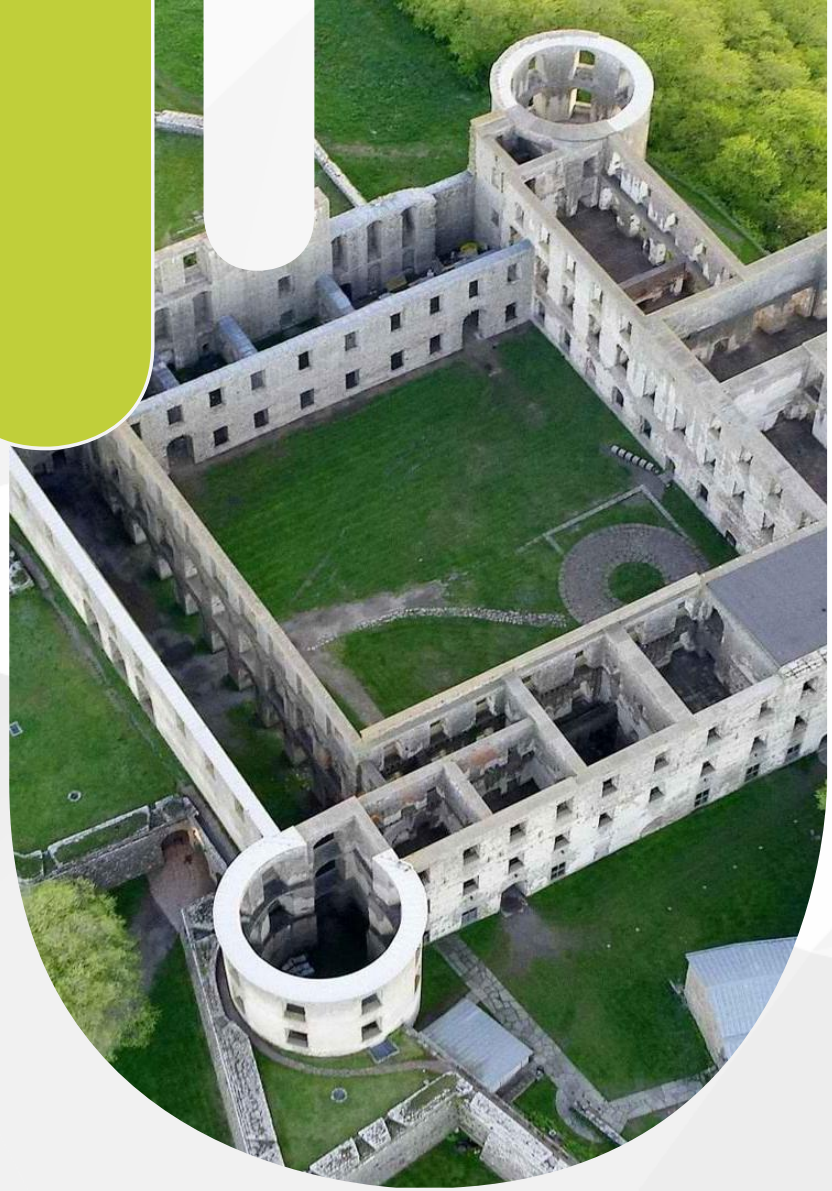
Ce style défensif ne remet pas en cause les bonnes pratiques du développement objet :

- Les principes et les règles du développement sécurisé contribuent aussi à la qualité du code et sont compatibles avec le développement objet.
- 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
- Injection de code
- Données choisies pour ralentir les algorithmes

Privilégier la défense plutôt que l'attaque



LA DEFENSE

La défense est fondamentalement plus difficile que l'attaque

- La programmation défensive repose sur des principes stables.
- Rendre robuste un système d'information passe donc nécessairement par une action au niveau du logiciel, dès son développement.
- Pour présenter ces principes généraux, nous nous appuierons sur la comparaison avec l'architecture militaire, dans les châteaux-forts médiévaux ou les forts Vauban.

Les principes du développement sécurisé

Les lignes de défense : fortifications, fossés, tours etc.

- En logiciel, les différents composants logiciels 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 (pont-levis, herse..)

- Avec des contrôles sur les accès (vérification des entrants, mots de passe etc.).
- En logiciel, un système d'identification des utilisateurs habilités à se servir du logiciel.

Les principes du développement sécurisé

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.
- 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 (WEB et BDD).

Le rempart extérieur : des éléments de défense les plus indépendants possibles

- 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.

Les principes du développement sécurisé

Minimiser le périmètre de l'application

- KISS : Plus le code restera petit et simple, plus il sera ensuite facile de vérifier la sécurité de l'application.
- Développer uniquement les fonctionnalités nécessaires et suffisantes YAGNI
  - Utilisation du cycle en V : on ne développe que ce qui est demandé dans le cahier des charges.
  - Utilisation du cycle itératif des méthodes agiles : au fur et à mesure des itérations, on développe ce qui demandé en chaque itération.

Privilégier la simplicité

- Diviser les fonctionnalités complexes en fonctionnalités courtes, simples et ciblées
  - Chaque classe ou méthode fait peu de choses et le fait bien.
  - Facile à comprendre, à documenter et à maintenir

Les principes du développement sécurisé

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.
- Principe du pattern de conception objet de faible couplage afin de limiter le plus possible les dépendances entre les classes.
    - Avec un couplage fort :
        - Si l'un des éléments est compromis alors toute la chaîne sera compromise.
        - Si je viens à modifier une dépendance, celle-ci ne doit pas avoir d'impacts sur la chaîne de dépendance au risque de devoir modifier toute la chaîne.

Factoriser le code

- Factorisation intelligente : une méthode est appelée dans plusieurs contextes, pour remplir réellement la même fonctionnalité.
    - Clarté et simplicité
- 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).
    - Le risque est que le hacker va jouer sur le paramètre de contrôle pour exécuter la fonctionnalité de B alors qu'il n'a le droit qu'à A

Les principes du développement sécurisé

réutiliser intelligemment le code

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

Utiliser l'encapsulation

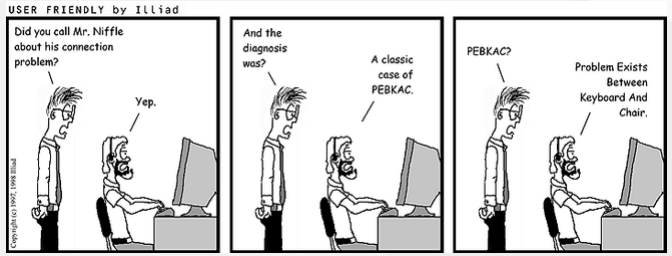
- L'encapsulation ne joue pas sur la taille du logiciel mais sur sa simplicité, par le mécanisme d'abstraction : l'utilisateur d'une classe n'a pas à voir et ne verra pas les détails d'implémentation de cette classe qui sont cachés (encapsulés, annotés private).
- Il ne voit que l'essentiel de la classe,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

Adopter une posture de méfiance

Adopter une posture de méfiance

Les données externes

- Pour un logiciel, il faut se préoccuper :
  - Des saisies utilisateurs
    - Buffer overflow, injections SQL, CSRF
  - Trames réseaux (DDOS)
  - Cohérences des données provenant d'un fichier, d'une base de données externes, d'une API etc…
- On doit se méfier de tout ce qui provient de l'extérieur à l'application même si une confiance existe entre les composants.



Les entrées / sorties et les interfaces (UI)

- 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.

Adopter une posture de méfiance

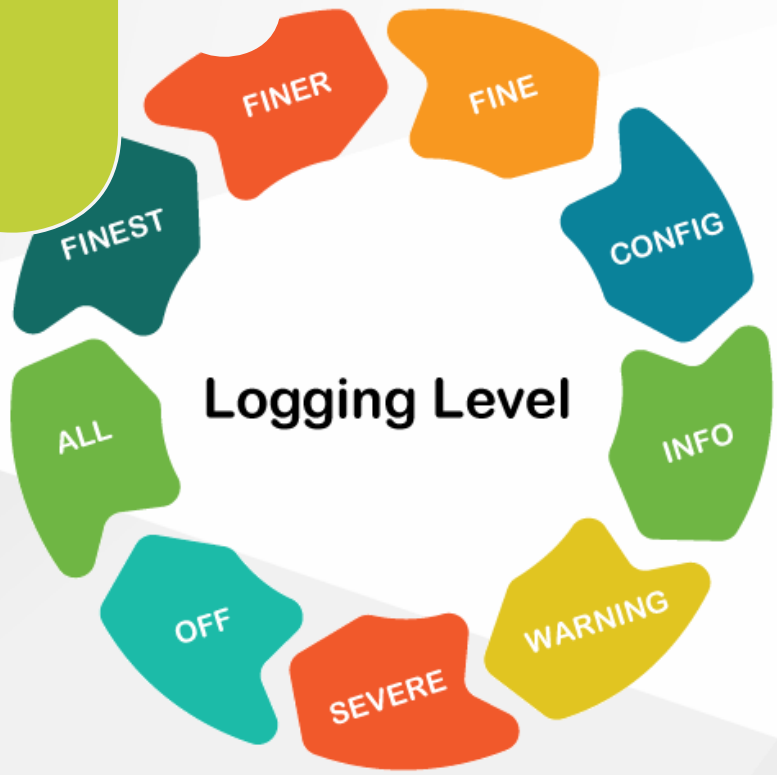
DEFENSE EN PROFONDEUR (DEFENSE IN DEPTH)

- Limiter sa confiance envers l'extérieur. Il faut :
  - ne prêter aucune confiance aux données externes, qui doivent être soigneusement validées.
  - 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
  - Exemple 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.

SEPARER ET MINIMISER LES PERMISSIONS ET LES PRIVILEGES

- 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 un mot de passe que l'on change fréquemment
  - privilèges séparés et minimum : chaque individu n'a le droit d'exécuter que le minimum d'actions qui correspondent à sa fonction.
  - permissions séparées et minimales : chaque individu n'a accès qu'au minimum de ressources exigées par sa fonction
- 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).

Journaliser



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.

Maitriser son langage et exploiter au mieux son IDE

LANGAGE et IDE

LANGAGE

- Les langages sont prévus pour des objectifs différents et sont très inégaux face à la sécurité (par exemple C et Java)
- Faire de la revue de code dont le but est de faire relire son code par un autre développeur afin de détecter des problèmes qu'on n'aurait pas détecter.
- Faire de la veille de sécurité permanente même après codage
  - Suivre les normes et guides de développement sécurisé. (CERT par exemple)
  - Utiliser les mécanismes de sécurité existants et robustes.
- Faire et ne pas négliger les phases de tests de son code.

IDE

- Pour un même langage, les compilateurs peuvent avoir un comportement différent selon les éditeurs de logiciel.
- 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.
  - Ils sont souvent accompagnés d'analyseur statique intégré afin de détecter les risques dans le code source.
  - Utiliser des analyseurs de code comme SonarCloud, SonarCube.

Les règles de programmation défensive en Java

https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

CERT sur 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,
- 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 coutera-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)

Exemple de règles

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.
- 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.

Solutions proposées par le CERT

- 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

Pour aller plus loin

FAITES VOTRE PROPRE VEILLE TECHNOLOGIQUE

Sur la sécurité en Java

- 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

Merci

afpa.fr

Jérôme BOEBION

version 1.0 - 2024