Notions de développement sécurisé

Table des matières

1 Principes de base

1.1 Liste de principes

Les principes suivants permettent de développer des applications sécurisées et qui seront utilisées de façon sécurisée.
Suivre le principe **KISS** : Keep It Simple & Stupid
Adopter un design ouvert. Eviter de créer son propre design, privilégier les designs existants en les comparant au cahier des charges du projet.
Vérifier constantemment les autorisations des utilisateurs. Les attaques peuvent provenir aussi bien de personnes externes que d'utilisateurs authentifiés. Il est déjà nécessaire de vérifier que pour toute action demandée par un utilisateur, celui-ci a l'autorisation de l'effectuer avant de l'exécuter.
Toujours choisir des défauts sécurisés. Dans de nombreux développements il est nécessaire de choisir quoi faire dans les cas par défaut si le cahier des charges ne le précise pas. Il est alors préférable de choisir le cas le plus sécurisé par défaut. Par exemple si un utilisateur n'est pas explicitement autorisé à exécuter une fonction, alors cela lui sera refusé.
Faire en sorte de faciliter l'utilisation de l'application pour empêcher l'utilisateur d'avoir envie de contourner la sécurité. Par exemple si l'utilisateur doit changer son mot de passe tous les mois alors il l'écrira sur un postit collé sur son écran.

2 Méthodologies appliquées

2.1 Filtrer les données en entrée

Une politique efficace dans un très grand nombre de cas est de filtrer en entrées toutes les données reçues de tiers de non confiance.
Ce filtrage doit être effectué dès la réception de la donnée et avant toute utilisation de celle-ci.
Le filtrage doit être composé de la vérification de la longueur de la donnés, avec des limites minimales et maximales ; et de la vérification des caractères utilisés. Un exemple classique est le code postal (français) qui doit être composé de 4 à 5 chiffres.
Lorsqu'une donnée reçue ne correspond pas à ce qui est attendu, alors un message d'erreur générique est généré et renvoyé à l'utilisateur, un numéro d'enregistrement étant joint. Le programme journalise localement l'erreur détaillée, accompagnée du numéro d'enregistrement envoyé à l'utilisateur.
Si une donnée obligatoire est absente alors une erreur est générée.
Si une donnée non attendue est reçue, la réaction dépendra de la politique choisie, soit d'ignorer totalement la donnée, soit de générer une erreur.
Quand une erreur est générée pendant le filtrage, la fonction appelée n'est pas exécutée et retourne une erreur.
Si le filtrage a validé toutes les conditions alors la fonction appelée peut enfin être exécutée.
Le filtrage des données peut être effectué de différentes façons, mais l'utilisation d'expressions rationnelles (regular expressions) est l'une des méthodes les plus simples.

2.2 Encoder les donnés en sortie

Dans certains cas il est nécessaire d'encoder les données en sortie. Ceci est nécessaire à chaque fois que le client qui présente les données à l'utilisateur interprète les données reçues et que celles-ci peuvent contenir du code exécutable, par exemples les données envoyées à un navigateur html qui pourrait contenir du javascript, ou celles envoyées à une base SQL qui pourrait contenir du code SQL.
Les fonctions spécifiques d'encodage sont alors à utiliser suivant le contexte : navigateur WEB, serveur SQL…

2.3 Minimiser les privilèges

Si le programme a besoin de privilèges alors mettre en place le principe de moindre privilège, en minimisant le temps où les privilèges sont accessibles, et de séparation des privilèges si besoin.

2.4 Vérifications des valeurs retour

Un bon principe est de vérifier que toutes les valeurs retournées par les fonctions appelées correspondent aux valeurs attendues.
Des fonctions comme **strlcat** retournent la taille des données traitées, si cette taille n'est pas celle de la chaîne source alors un problème a eu lieu.
Une autre exemple est les fonctions de restriction de privilèges telles que **chroot()**, **setreuid()**. Si une erreur est retournée cela signifie que la restriction des privilège n'a peut être pas fonctionné et que du code de non confiance, ou traitant des données de non confiance, va se faire dans un contexte privilégié.

2.5 Sécurité par l'obscurité

La sécurité par l'obscurité est le fait de tenter de garder secrètes des informations sur la structure, le fonctionnement ou l'implémentation d'objets ou d'un procédé, afin de tenter d'assurer la sécurité de ces objets ou procédés.
Historiquement des donnés pouvaient être « protégées » par l'algorithme **ROT13** qui décale de 13 caractères dans l'alphabet les lettre d'un message. Cet algorithme, comme d'autres étant connus et n'ayant aucune clé de chiffrement, les données protégées peuvent être dévoilées instantanément.
Dans certains cas l'algorithme n'était pas connu, mais via des analyses basiques de génération de textes protégés à partir de textes en clair il est possible de retrouver l'algorithme de protection et d'en déduire l'algorithme de dé-protection.
Un exemple est la sécurité par l'obfuscation où des données normalement accessibles et compréhensibles sont rendues difficilement compréhensibles par l'humain mais sans changer quoi que se soit à son interprétation sur ordinateur. Par exemple des programmes permettent de réécrire des sources de langages tels que **javascript**, **perl** ou autres interprétés tels quels, ou les byte codes de langages tels que **java** qui sont interprétés par la machine virtuelle.

2.6 Limitation des erreurs retournées à l'utilisateur

Lors d'erreurs, les messages envoyées à l'utilisateur ne doivent pas contenir d'informations techniques sur la cause de l'erreur.
Cacher ces données rend plus complexe la mise au point de l'exploitation d'une faille car l'attaquant ne sait pas comment modifier ses données malveillantes afin que son exploitation foctionne.
Par exemple une requête à une base de données peut renvoyer des informations techniques sur la requête et/ou la base de données. Ces données pourraient permettre à l'utilisateur de corriger les données malveillantes qu'il envoie au programme afin d'exploiter la faille qu'il a identifié.
D'autres exemples est lors de l'authentification de ne pas signaler lequel du login ou du mot de passe est erroné, l'attaquant ne sais alors pas si le login testé existe ou pas.

2.7 Effacement des données sensibles en mémoire

Sur tous les systèmes, quand un pirate a réussi a pirater le système, toutes les données hébergées par celui-ci peuvent être volées, aussi bien les données enregistrées sur le disque dur que dans la mémoire des processus.
Quand un programme utilise une donnée sensible, par exemple un mot de passe ou une clé secrète, il est nécessaire que le temps d'utilisation soit minimal et que la mémoire soit ensuite nettoyée de cette donnée.
En C il est nécessaire d'écraser un à un tous les caractères de la donnée avec une donnée neutre. Changer le premier caractère par le caractère nul de fin de chaîne ne suffit pas puisque les autres caractères se trouvent toujours en mémoire.
Il est aussi possible que le processus soit déplacé en swap (fichier d'échange). Si cela arrive alors la donnée sensible se retrouvera écrite sur le disque dur jusqu'à ce qu'elle soit remplacée par une autre donnée, ce qui peut être bien après que le programme ait nettoyé sa mémoire et ait même mis fin à son exécution.

Pour protéger la donnée sensible en mémoire il est nécessaire d'utiliser la fonction **mlock()** sur l'espace mémoire où la donnée sera chargée par la suite. Après nettoyage de cette donnée en mémoire, il est possible d'utiliser **munlock()** pour déverrouiller cet emplacement mémoire.

2.8 Algorithmes et protocoles cryptographiques

Il est déconseillé de développer soit même des fonctions de chiffrement ou de hachage, et encore moins de créer soit même ses propres algorithmes. La cryptographie est une matière extrêmement complexe et les erreurs durant la conception ou le développement sont systématiquement catastrophiques pour la sécurité des données.
Il existe de nombreuses bibliothèques de confiance telles que **OpenSSL**, **LibreSSL** et **GnuTLS**. Attention à utiliser ces bibliothèques en suivant les règles décrites dans les documentations respectives car ici aussi une mauvaise utilisation pourra être catastrophique pour la sécurité des données.

2.9 Nombres aléatoires

Il est déconseillé de développer soit même des fonctions de génération de nombres aléatoires.
Sur tous les systèmes de type **Unix**, les interfaces **/dev/random** et **/dev/urandom** fournissent des nombres aléatoires de qualité, alimentés par le temps aléatoire entre deux événements matériels.
Sur les autres systèmes, utiliser les fonctions de génération de nombres aléatoires des bibliothèques de chiffrement sus-citées.

2.10 Logiciels et bibliothèques à jour

Il est important que les logiciels et bibliothèques utilisés par votre programme soient à jour. En effet ceux-ci peuvent contenir des failles et peuvent être exploités via votre programme s'il utilise des fonctions vulnérables.
Par exemples pour une application **web**, celle-ci sera vulnérable si **PHP** ou une bibliothèque telle que **XML**… sont vulnérables. Pour une application **C/C++**, celle-ci sera vulnérable si une bibliothèque telle que **Qt**, **libc**… sont vulnérables.

2.11 Droits d'accès sécurisés sur fichiers du programme

Il est important d'avoir des droits d'accès sécurisés sur les fichiers du programme. Les utilisateurs d'un serveur ne doivent pas pouvoir modifier les fichiers (exécutables, bibliothèques, configuration, données…) de votre programme, seul l'administrateur doit pouvoir modifier ceux-ci.
Si certains fichiers contiennent des données sensibles, comme des mots de passe d'accès à une base de données, il ne faut pas que celles-ci soient accessibles aux utilisateurs. Dans une application **WEB**, ces fichiers sont positionnés en dehors de l'arborescence **HTTP** afin qu'ils ne soient pas accessibles via **HTTP**.

3 Cycle de vie

3.1 Conception

La sécurité doit être prise en compte dès la conception de l'application.
Dans de plus en plus d'entreprises une politique de sécurité est appliquée et doit donc être appliquée dans toutes les applications, même celles qui sont développées par et/ou pour l'entreprise.
La politique de sécurité peut influencer l'architecture des applications, par exemple il peut être obligatoire de séparer les serveurs **WEB** et les serveurs **SQL**, et de positionner un relais inverse devant les serveurs **WEB** afin de filtrer les requêtes mal formées.
La politique de sécurité peut également obliger :

- à limiter les privilèges utilisables par les applications sur les systèmes,
- à restreindre les connexions vers les serveurs sensibles…

3.2 Politique de développement

La politique de sécurité peut comporter une politique de développement, influençant le cycle de développement de l'application.
La politique peut obliger :

- à avoir un gestionnaire de versions de sources,
- à effectuer des audits de code,
- à former les développeurs au développement sécurisé,
- à effectuer des tests d'intrusion contre les applications…

3.3 Tests

Afin de vérifier que le développement ne cache pas des bugs, des tests sont généralement mis en place. Souvent ces tests ne font que vérifier que l'application fonctionne correctement dans les cas optimaux.
D'un point de vue sécurité il est intéressant de vérifier la réaction du programme avec :

- des données mal formatées ou
- des cas d'utilisation non prévus lors de la conception.

Des exemples sont :

- des chaînes trop longues,
- des données aberrantes (des lettres à la place de chiffres…),
- des valeurs aux limites (0 pour un nombre strictement positif, 128 pour un octet signé, un pointeur null ou avec une valeur non possible)…

S'il est surtout utilisé contre les applications **WEB**, toutes les applications peuvent être testées avec du **fuzzing**. Il s'agit à partir de requêtes valides, d'en générer en appliquant des modifications plus ou moins aléatoires et de voir le comportement de l'application.

3.4 Audits de code

Un audit de code permet de trouver dans le code source d'une application des failles qui ont été insérées par les développeurs. Il est généralement mené par des personnes externes à l'équipe de développement.
Il existe deux types d'audits de codes : statiques et dynamiques. Dans un audit statique, le code est analysé, par un logiciel et ou un auditeur. Dans un audit dynamique, l'exécution du code est simulée par un logiciel et certaines problématiques peuvent apparaitre plus facilement que seulement avec un audit statique.
Dans les deux cas l'utilisation d'un logiciel automatique génère de nombreux faux positifs qu'il faut ensuite analyser manuellement.

3.5 Tests d'intrusions

Un test d'intrusion permet de tester les réactions d'une application contre un utilisateur malveillant. Il est généralement mené par des personnes externes à l'équipe de développement.
Il sagit d'un excellent complément à l'audit de code, certains points ne pouvant pas être vérifiés par un audit de code, comme la configuration des services, les versions des services (si elle est vulnérable ou pas), ou certaines réactions à certaines attaques, comme les dénis de service.

3.6 Diffusions

Lors de la diffusion du logiciel, l'intégrité de celui-ci devra être assurée. Ceci peut être effectué via la génération d'une signature cryptographique.
Les systèmes permettant de générer le logiciel doivent également être vérifiés contre l'infection par des logiciels malveillants qui pourraient contaminer le logiciel.

3.7 Avis de sécurité et mises à jour de sécurité

Lorsqu'une faille de sécurité est trouvée sur le logiciel, il est nécessaire de suivre une procédure pré-établie.
D'abord analyser la faille pour la corriger.
Ensuite analyser la faille pour déterminer la gravité de la faille.
Puis mettre à disposition rapidement un patch de sécurité permettant de corriger le problème sur toutes les versions vulnérables.
Enfin diffuser dans une liste de diffusion dédiée à la sécurité de vos produits un avis de sécurité mentionnant :

- le type de faille,
- la conséquence,
- les versions impactées et
- l'endroit où les patches de sécurité sont disponibles.

Toutes ces étapes doivent être réalisées rapidement, généralement en quelques jours, et cela d'autant plus rapidement que les conséquences de l'exploitation de la faille sont graves.