La sécurité dans les systèmes d'exploitations

Ce cours présente un aperçu sur le sujet de la sécurité dans les systèmes d'exploitation. Après avoir présenté quelques concepts généraux, nous nous intéressons aux mécanismes de sécurité utilisés dans les systèmes UNIX.

C'est un sujet très important:

- Les applications s'exécutent au dessus de l'OS
- Si la sécurité de l'OS est compromise, c'est le cas aussi pour les explications qui l'utilisent

1 Définition de la sécurité

Des propriétés qui définissent ce que nous aimerions qu'il se passe et ce que nous ne voulons pas qu'il se passe.

- De plus, nous voulons des garanties fortes que ces propriétés soient correctement mises en oeuvre

1.1 3 grands objectifs

- Confidentialité: Empêcher l'accès à des données à un utilisateur non autorisé
- Intégrité: Empêcher un utilisateur d'altérer (voir supprimer) des données pour lesquelles il n'est pas autorisé à le faire
- Disponibilité: Éviter qu'un adversaire puisse empêcher l'accès à un service pour les autres utilisateurs

2 Les grands principes

2.1 Mécanismes de base

- Isolation
  - Isolation des processus applicatifs entre eux
  - Isolation de l'OS par rapport aux applications
  - On note que quand on parle d'isolation, il faut se poser la question de l'accès au matériel puisque tous les processus et l'OS se partagent l'accès au matériel.
- Authentification des utilisateurs
  - Des droits associés à chaque utilisateur et donc, à chaque processus appartenant à l'utilisateur

2.2 Principes généraux

- Trusted Computing Base (TCB) à minimiser
  - La quantité de code dans laquelle on doit avoir confiance doit être minimisée
    - Moins de risques de bugs
      - Entre 1 et 6 bugs pour 1000 lignes de code (pour de bon programmeurs)
    - Réduction de la surface d'attaque
    - Plus facile de vérifier que le code est correct
      - Automatiquement ou manuellement
  - Idée de *minimisation des mécanismes partagés*
- Les mûrs entre les domaines isolés doivent être aussi hauts que possibles
  - Ils doivent avoir le moins possible de mécanismes en commun
  - Les moyens d'interagir entre domaines doivent être rigoureusement contrôlés
- Interdiction par défaut

  - Les mécanismes de sécurité doivent interdire tout ce qui n'a pas été explicitement autorisé
- *Open Design*: N'importe qui doit pouvoir étudier le design de l'OS
  - Si la sécurité est fondée sur le secret, très grand danger si le secret ne peut plus être garanti
  - Si tout le monde a accès, les bugs et les vulnérabilités ont plus de chance d'être trouvés (et corrigés)

3 Mise en oeuvre dans les systèmes UNIX

Dans le suite, nous nous concentrons sur les mécanismes mis en oeuvre au sein des systèmes UNIX.

3.1 Authentification des utilisateurs

3.1.1 Le démarrage du système

Après que les premières phases du *boot* sont passées (configuration du matériel, lecture sur le disque des informations concernant l'OS à démarrer, démarrage du noyau), les étapes suivantes sont exécutées:

- Le processus `init` est démarré (le *père* de tous les processus)
- Le processus `init` fork un nouveau processus qui va exécuter le programme `/bin/login`

3.1.2 Le programme /bin/login

- Demande le nom d'utilisateur et le mot de passe
- Chiffre le mot de passe
- Vérifie que le mot de passe chiffré correspond au mot de passe chiffré stocké dans le fichier `/etc/passwd`
- Si oui:
  - Le programme `/bin/login` utilise `exec()` pour charger un shell dont le propriétaire est l'utilisateur.

3.1.3 Contrôle de l'accès aux ressources

UID et droits d'accès

- Chaque utilisateur est identifié par un identifiant unique: `UID`
- A chaque processus est associé l'UID de l'utilisateur qui l'a créé
  - (Le processus appartient aussi au groupe de l'utilisateur)
  - Les processus fils d'un processus héritent de son UID
- Les droits d'accès permettent de définir quels utilisateurs, et donc quels processus, peuvent accéder à chaque donnée stockée sur le système

3.2 Isolation

3.2.1 Isolation des processus

Question: Comment assurer qu'un processus n'accède pas aux données en mémoire d'un autre processus?

La mémoire virtuelle

- L'OS offre un espace d'adressage virtuel privé pour chaque processus
  - Les mécanismes permettant de traduire les adresses virtuelles en adresses physiques sont mis en oeuvre par l'OS en collaboration avec le matériel
- Description simplifié du fonctionnement:
  - Quand un processus écrit une donnée dans sa mémoire virtuelle, l'OS choisit à quel endroit dans la mémoire physique la donnée va être stockée
  - L'OS stocke ensuite dans une structure de données dédiée, stockée en mémoire, la correspondance adresse virtuelle et adresse physique
    - Cette structure s'appelle la *table des pages*
    - Une table des pages par processus
    - Le processeur sait lire la table des pages
      - Quand un processus s'exécute sur le processeur, un registre du processeur pointe vers le début de la table des pages du processus

        - Quand le processeur doit exécuter une instruction de lecture/écriture, il accède à la table des pages du processus pour faire la traduction de l'adresse virtuelle en adresse physique
- Grâce à ce mécanisme, un processus ne peut pas accéder aux adresses en mémoire qui ne stockent pas des données qui lui appartiennent

Context switch *Pour simplifier, nous supposons un système avec un seul processeur (et un seul coeur)*

- Dans le système, il y a en général plusieurs processus prêts à s'exécuter
  - L'OS doit donner accès au processeur à chacun de ces processus de manière alternée
- Quand l'OS décide d'exécuter le processus `P2` à la place du processus `P1` (`context switch`), comment éviter que `P2` n'accède à des données appartenant à `P1`?
  - L'OS arrête `P1` et sauvegarde son état
    - Sauvegarde des registres du processeurs (`program counter`, pointeur vers la table des pages, etc.) dans un endroit en mémoire inaccessible pour les autres processus
  - L'OS charge l'état de `P2` dans le processeur
    - Ceci inclut la mise à jour du pointeur vers la table des pages. On pointe maintenant vers la table des pages de P2
  - A partir de cette instant, toutes les adresses virtuelles sont traduites en adresse physiques correspondant aux données de `P2`

3.2.2 Isolation du système d'exploitation

Les domaines de sécurité

- Dans les premiers systèmes d'exploitation, il n'y avait qu'un seul domaine de sécurité
  - Les applications et l'OS faisaient partie du même domaine de sécurité
    - Une application pouvait *facilement* compromettre le fonctionnement du système d'exploitation
  - TCB très grand
    - Inclus toutes les applications
- Dans les systèmes UNIX, il y a 2 domaines de sécurité
  - Un domaine pour l'OS (*kernel mode*)
  - Un domaine pour les applications (*user mode*)
  - Certaines designs de systèmes d'exploitations utilisent plus de domaines (compromis entre efficacité et sécurité)
- Comment éviter que les applications des utilisateurs exécutent des opérations qui peuvent compromettre le bon fonctionnement et la sécurité du système
  - Certaines opérations ne doivent pouvoir être exécutées que en *kernel mode*
  - Pour assurer cela, nous avons besoin d'un support matériel

Les niveaux de privilège du processeur Les processeurs modernes mettent en oeuvre différents niveau de privilège:

- Le niveau 0 est le niveau avec le plus de privilège: il permet d'exécuter n'importe quelle instruction
- Les niveaux supérieurs ont moins de droits:
  - Ils ne permettent pas d'exécuter certaines instructions
  - Par exemple, ils ne permettent d'exécuter des instructions de lecture et d'écriture vers les périphériques de stockage
  - Dans UNIX, le *user mode* correspond typiquement au niveau 3 sur les processeurs Intel
- Le niveau utilisé est stocké dans un registre du processeur

Passage d'un mode à l'autre (mode switch) – les appels systèmes Ce principe de fonctionnement, fondé sur 2 modes d'exécution, implique que les applications doivent demander l'aide du système d'exploitation pour exécuter certaines opérations. Ceci est fait au travers *d'appels systèmes*.

Voici les principes de fonctionnement des appels systèmes (en quelques mots):

- Un processus exécute un programme (par exemple codé en C), qui fait un appel système (par exemple `read()`):
- Ceci appelle la fonction de la bibliothèque C (`read()`) qui effectue les opérations suivantes
  - Elle écrit dans des registres les informations à propos des paramètres de l'appel système

  - Elle exécuté une instruction `trap`
    * Instruction qui sert à envoyer une interruption au matériel
- Sur réception de l'interruption, le processeur:
  - Modifie le registre permettant de passer du *user mode* au *kernel mode*
  - Appelle la fonction du noyau qui met en oeuvre l'appel système demandé
- A partir de ce moment, l'OS:
  - Peut vérifier les paramètres de l'appel système pour voir si celui-ci peut s'exécuter
  - Si oui, il exécute l'opération pour le compte du processus utilisateur