TP n° 10: Liaisons dangereuses

Ce TP a pour but d'expliquer la différence entre la liaison statique et dynamique à la compilation, et entre autre pourquoi lors du TP n° 4, nous avons utilisé une liaison statique et non dynamique…

Exercice 1. *Bibliothèques statiques*

Lorsque vous écrivez un programme, vous pouvez avoir besoin de fonctionnalités qui ont déjà été implémentées par une autre personne. Il est souvent plus pratique de ne pas réinventer la roue et de faire appel à ces fonctionnalités au lieu de les re-implémenter.

1. Créez votre bibliothèque "extra simpliste" :
   Dans un fichier `carre.c`, écrivez une fonction `carre` qui prend en argument un entier n et qui renvoie n^2.
   Créez aussi un fichier `carre.h` contenant uniquement la définition de cette fonction.
2. Compilez le fichier `carre.c` dans une bibliothèque statique avec les commandes suivantes :

   ```
   gcc -c carre.c -o carre.o
   ar rcs libcarre.a carre.o
   ```

   À quoi correspond l'option `-c` du `gcc` ? Que fait la commande `ar` ?
   Déduisez-en qu'est-ce qu'une bibliothèque statique.
3. Dans un fichier `main.c`, écrivez un programme C qui affiche (avec `printf`) le carré du nombre n passé en argument sur la ligne de commande. Vous devez utiliser la fonction `carre` (donc inclure son header), mais vous ne devez pas inclure le code de la fonction.
4. Compilez votre fichier ainsi :

   ```
   gcc main.c -L. -lcarre -o main_lib_statique
   ```

   Utilisez le manuel de `gcc` pour démystifier les options `-L` et `-l`.
5. Dans votre programme vous avez utilisé la fonction `printf` pour l'affichage et vous avez certainement inclus `<stdio.h>` dans `main.c`. Comment le compilateur connaît la fonction ?
   Pas de magie : retrouvez la définition de la fonction `printf` dans le fichier `/usr/include/stdio.h`
6. Utilisez la commande `objdump` pour analyser le fichier `main_lib_statique` produit par le compilateur (`objdump -S main_lib_statique`).
   Où se trouve le code assembleur du `main` ? et de la fonction `carre` ? et du `printf` ? En déduire la différence entre une bibliothèque statique et une bibliothèque dynamique.

Exercice 2. *Liaison dynamique*

L'implémentation des fonctions externes est souvent faite dans une bibliothèque dynamique (l'extension `.so` dans Linux ; `DLL` sur windows). Lorsque vous compilez votre code, le compilateur indique dans l'exécutable généré qu'avant de lancer le programme, une bibliothèque doit être chargée en mémoire, c'est la liaison dynamique : la liaison entre le programme et la bibliothèque est faite à l'exécution par `ld.so` (voir le manuel).

1. Compilez votre bibliothèque dynamique :

   ```
   gcc -shared carre.c -o libcarre.so
   ```

   Compilez votre programme principal : `gcc main.c -o main_lib_dynamique -L. -lcarre`.
2. Que se passe-t-il quand vous lancez le binaire `./main_lib_dynamique` ? Comment régler ce problème ?

Exercice 3. *Tout est dépeuplé*

Au moment du lancement d'un exécutable faisant appel aux fonctions des bibliothèques dynamiques, `ld.so` cherchera les fonctions dont il a besoin dans une liste de bibliothèques présentes sur votre machine.
Cette recherche est *ordonnée*. Cela signifie que plusieurs bibliothèques peuvent implémenter la même fonction et la première trouvée sera utilisée. La liste ordonnée des bibliothèques peut être vue avec `ldconfig -p`.
Une variable d'environnement, `LD_PRELOAD` permet de rajouter des bibliothèques en tête de la liste. Cela permet donc de redéfinir une fonction existante.

1. Écrivez un fichier C `malefique.c` qui contient une fonction `readdir` qui a le même prototype que la fonction POSIX que vous connaissez, mais qui renvoie toujours `NULL`.

   Compilez ce fichier en `libmalefique.so`.
2. Lancez la commande `LD_PRELOAD=$PWD/libmalefique.so ls`. Expliquez le comportement.
3. Téléchargez le `ls` fourni avec le sujet dans le répertoire "Partie2". Répétez la commande précédente avec ce `ls`. Expliquez ce qu'il se passe.
4. Modifier `malefique.c` pour intercepter l'appel à `read` et donner l'impression que les fichiers sont vides.

Exercice 4. *Kernel hacker*

Cet exercice est **optionnel**. Le but est de commencer à comprendre le fonctionnement des modules noyaux. **Attention :** Vous avez besoin d'être root sur la machine pour cet exercice, de plus vous devez savoir ce que vous faites, un bug dans un module noyau peut amener à des conséquences *sympathiques*...
C'est pourquoi le code ainsi que le Makefile sont fournis, afin d'éviter des problèmes. Si vous n'avez pas les sources du noyaux vous devez les installer, usuellement le paquet se nomme `linux-headers`.

1. Compilez votre premier module noyau avec `make`. L'extension du module est `ko`.
2. Chargez votre module dans le noyau à l'aide de la commande `insmod`. Vérifiez à l'aide `lsmod` que ce module est bien chargé.
3. Que fait ce module ?
4. Déchargez votre module dans le noyau à l'aide de la commande `rmmod`.