TP/TD 6 : Un rhume logiciel

1 Les bibliothèques statiques et dynamiques

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

Question 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.

Question 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
```

A quoi correspond l'option `-c` du `gcc` ? Que fait la commande `ar` ? Déduisez-en qu'est-ce qu'une bibliothèque statique.

Question 3

Dans un fichier `main.c`, écrivez un programme C qui affiche (avec `printf`) le carré du nombre de ses arguments sur la ligne de commande en utilisant la fonction `carre`. Vous devez bien faire appel à la fonction `carre`, mais vous ne devez PAS inclure cette fonction dans le fichier `main.c`.

Note : vous devez faire un `"include"` du fichier `carre.h` dans `main.c`

Question 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`.

Question 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`

Question 6

Utilisez la commande `objdump` pour analyser le fichier `main_lib_statique` produit par le compilateur (`objdump -S main_lib_statique`). Comme alternative, nous vous fournissons le résultat d'un autre logiciel de désassemblage (dans le répertoire "Partie1/Dissam" fourni avec le sujet, c'est le fichier mainStatique.png). Nous vous demandons de concentrer votre attention sur les 3 parties du fichier qui sont marquées avec un rectangle rouge (vers 1/3 de l'image et toute à la fin de l'image).

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.

Note: Nous avons donné l'intégralité du désassemblage pour les personnes intéressées.

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 bibliotèque 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).

Question 7

Compilez votre bibliothèque dynamique :

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

Compilez votre programme principale: `gcc main.c -o main_lib_dynamique -L. -lcarre` .

Question 8

Que se passe-t-il quand vous lancez le binaire `./main_lib_dynamique` ? Comment régler ce problème ?

2 Une seule bibliothèque est chargée, et 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 re-définir une fonction existante.

Question 9

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`.

Question 10

Lancez la commande `LD_PRELOAD=$PWD/libmalefique.so ls`. Expliquez le comportement.

Question 11

Téléchargez le ls statique fourni avec le sujet dans le répertoire "Partie2". Répétez la commande précédente avec ce `ls`. Expliquez ce qu'il se passe.

Question 12

Modifier `malefique.c` pour intercepter l'appel à `read` et donner l'impression que les fichiers sont vides.

3 Encore plus maléfique

Les fichiers exécutables au format ELF utilisés par Linux suivent un format standardisé que vous pouvez voir dans la Figure 1. Dans la suite, nous allons analyser en détails ce format dans le but d'insérer un code maléfique dans un exécutable.

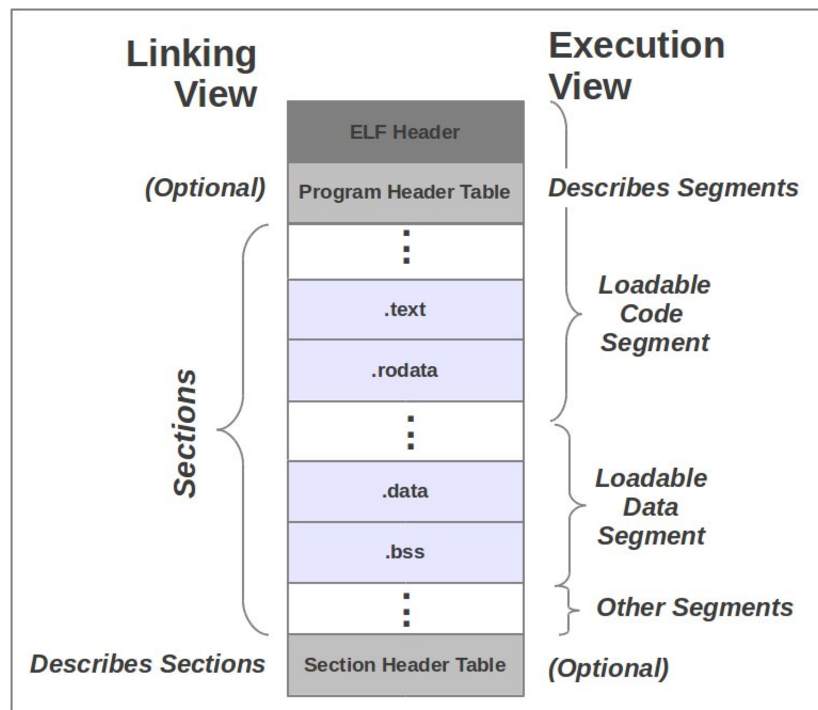

Figure 1: Un exécutable ELF

Question 13

Le code fourni se trouve dans le répertoire "Partie3". Compilez le fichier `main.c` avec la commande :

```
gcc  -static-libgcc -static main.c -o main_statique
```

Comparez la taille de l'exécutable avec ceux des exercices précédents. Commentez l'utilisation de l'option `-static-libgcc`.

Question 14

Exécutez `readelf -h main_statique` et analysez les informations stockées dans l'en-tête ELF. Notez bien l'adresse d'entrée du programme, car vous en aurez besoin par la suite.

Question 15

Analysez la sortie de la commande `readelf --section-headers main_statique`. En particulier, quels sont les `flags` des sections `.text`, `.rodata`, `.data` et `.comment` ? A quelle adresse mémoire seront chargées les données de la section `.rodata` ? Quelle est la taille de chaque section?

Question 16

Analysez la sortie du `readelf --program-headers main_statique`. Quelle est la différence entre une section et un segment ? Est-ce que l'OS s'opposera à l'exécution du code dans une adresse chargée depuis la section `.rodata`?

Note: Revenez sur l'image du désassemblage de la première partie du TD. Sur la colonne de gauche vous retrouverez les sections dont le chargement en mémoire a été simulé par le logiciel de désassemblage.

Question 17

En regardant le fichier **main.c** fourni, Vous avez probablement remarqué que nous avons réservé une section `".evilcode"` dans l'exécutable. Vous allez remplacer le contenu de cette section avec votre propre code écrit en assembleur.

Quelle est l'adresse mémoire à laquelle cette section sera chargée ? Notez-la.

Question 18

Récupérez le fichier "evil.S" fourni. Dedans, remplacez l'adresse de chargement du code assembleur (`org`) par l'adresse de la section `.evilcode`. Remplacez l'adresse de saut après l'exécution du code maléfique (`push`) par l'adresse initiale d'entrée dans le programme (celle que vous avez noté dans une question précédente).

Vous devez compiler le code assembleur vers un binaire en utilisant la commande `nasm` (dont binaire est fourni dans le répertoire "Partie3/binaries") : `./nasm -f bin -o evil.text evil.S`

Question 19

Utilisez la commande `objcopy` (binaire fourni avec le sujet) pour remplacer le contenu de la section `.evilcode` par le binaire que vous venez de compiler, et pour mettre à jour le point d'entrée dans le programme par l'adresse de cette section.

Question 20

Testez pour vérifier si votre code est bien exécuté avant le reste du programme.

Question 21

Bonus : L'erreur de segmentation à la fin du programme est suspecte, non ?

Bonus : Et s'il n'y avait pas de section réservée ?