TD – Pile d'exécution et tas

Nous analysons dans ce TD la façon dont sont organisés la pile d'exécution et le tas d'un programme écrit en C et exécuté sous Unix. L'environnement de référence est l'installation Kali Linux sous VirtualBox réalisée en début d'année, même si d'autres sont possibles.

Pour chaque exercice, vous devez concevoir, développer et exécuter un programme – de quelques lignes seulement, et le plus simple possible – qui permette de répondre aux questions posées. Les questions précédées de « (T) » n'appellent qu'une réponse textuelle, sans code.

Un point important est à noter : certaines des réponses dépendent de l'architecture de votre machine, Intel 64, IA-32 ou autre, et de votre environnement, notamment les limites mémoire (commande ulimit). Vous n'obtiendrez donc pas tous nécessairement les mêmes réponses.

Table des matières

1 Pile d'exécution

1.1 Position, sens de progression et taille maximale

(T) Quelle est la fonction d'un programme dont la zone de travail est au fond de la pile ?

Quelle est l'adresse mémoire (approximative) du fond de la pile d'exécution ?

Cette adresse est-elle la même à chaque exécution ?

(T) Pour quelle raison ?

Pour la suite du TD, voir les instructions en annexe.

Dans quel sens des adresses mémoire la pile progresse-t-elle ?

Quelle est la taille maximale (approximative) de la pile dans votre environnement ?

1.2 Position des variables locales et des paramètres

Où se trouvent les variables locales d'une fonction par rapport à ses paramètres ? Pour simplifier, prenez l'exemple d'une fonction avec un seul paramètre et une seule variable locale.

Combien d'octets les séparent ?

A quoi correspondent ces octets ?

Développez une fonction qui les modifie par erreur. Que se passe-t-il ?

1.3 Modes de passage des paramètres

Partie 1

Soit le programme suivant :

```
#include <stdio.h>

void fonction(int parametre) {
    printf("p = %d\n", parametre);
}

int main(int argc, const char* argv[]) {
    int variable = 1;
    fonction(variable);
    return 0;
}
```

Complétez ce programme pour répondre aux questions suivantes.

Montrez que `parametre` et `variable` sont bien deux zones mémoires distinctes, bien qu'à la ligne 7 du programme ce soit `variable` qui est passée en paramètre de `fonction`.

Montrer que l'on peut modifier `parametre` dans `fonction`, et que cette modification n'est pas répercutée sur `variable`.

(T) Comment appelle-t-on ce mode de passage de paramètre ?

Partie 2

On souhaite maintenant créer une nouvelle version de `fonction`, appelée `fonction2`, qui puisse modifier `variable`.

(T) Comment faire ?

Ajoutez `fonction2` au programme ci-dessus et testez-la.

(T) Comment appelle-t-on ce second mode de passage de paramètre ?

Au lieu de passer directement l'adresse de `variable` à `fonction2`, stockez cette adresse dans un pointeur, que vous déclarez dans `main`, puis passez ce pointeur en argument à `fonction2`.

Comme dans la partie 1 de l'exercice, montrer que le paramètre de `fonction2` et le pointeur déclaré dans main sont deux zones de mémoire distinctes.

(T) Par quel mécanisme le pointeur est-il passé à `fonction2` ?

(T) Pour un certain type d'arguments, il est possible d'utiliser le mode de passage de paramètre de la partie 1, alors qu'il est impossible d'utiliser celui de la partie 2. Quel est ce type d'arguments, et pourquoi ?

1.4 Nombre variable de paramètres

Certaines fonctions admettent un nombre variable de paramètres : on peut, par exemple, leur passer 3 arguments lors d'un appel, et seulement 2 lors du suivant. Ces fonctions sont dites *variadiques*.

(T) Citez les fonctions variadiques que vous connaissez (et constatez l'utilité des fonctions variadiques).

Consultez leur déclarations dans le man et répondez aux questions suivantes.

(T) Comment déclare-t-on en C des paramètres en nombre variables ? Vérifiez dans les fichiers `include` correspondants.

(T) Les paramètres en nombre variable doivent-ils tous avoir le même type ?

Répondez aux questions suivantes en faisant appel à votre intuition et en justifiant vos réponses.

(T) Une fonction peut-elle n'avoir que des arguments en nombre variable ?

(T) La déclaration d'une liste de paramètres en nombre variable peut-elle être suivie de la déclaration d'un paramètre classique ?

(T) En conséquence, où sont déclarés les paramètres en nombre variable dans la liste des paramètres : au début, au milieu, à la fin ou n'importe où ? Où sont-ils placés dans la pile d'exécution ?

Consultez l'article man 3 va_arg pour comprendre comment parcourir les paramètres en nombre variable dans la fonction appelée.

Développez une fonction qui retourne le plus grand d'un nombre quelconque de réels positifs ou nuls passés en paramètres.

(T) Quelle est l'erreur que le code appelant votre fonction peut commettre ?

(T) Sera-t-elle détectée par le compilateur ? Pourquoi ?

Vérifiez vos réponses en exécutant votre code.

2 Tas

Sur le modèle du paragraphe 1, répondez aux questions suivantes à l'aide d'un ou plusieurs programmes, les plus simples possibles, que vous développez.

A quelle adresse (approximative) se situe le tas ?

Quelle est la taille maximale (approximative) du tas dans votre environnement ?

Allouez successivement des blocs de 1, 2, 4 et 8 octets. Que constatez-vous sur la valeur des adresses retournées ?

Un bloc alloué (malloc) puis restitué (free) est-il réalloué à l'identique la fois suivante ?

Que se passe-t-il si votre programme écrit en dehors des limites d'un bloc qu'il s'est alloué ?

3 Annexe

Désactivation de l'ASLR. Voici deux méthodes parmi d'autres :

1- Lancer *une* exécution d'un programme avec l'ASLR désactivé :

```
$ setarch -R /chemin/vers/le/programme
```

2- Désactiver complètement l'ASLR jusqu'au prochain reboot du système :

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Attention, désactiver l'ASLR rend votre système vulnérable. N'oubliez pas de le réactiver après vos tests.