TP nº 4: Grande Débâcle Binaire

Le but de ce TP est de vous familiariser avec l'outil *gdb* (pour *GNU DeBugger*) que vous avez vu en cours dans le but d'analyser des programmes dont vous ne disposez que du binaire. Il s'agit d'une sorte de *capture the flag* dont le but sera de nous rendre un petit `rapport.pdf` avec les méthodes que vous avez utilisées pour venir à bout des exercices, et du message caché dans chaque programme.
Le rendu est à faire pour le vendredi 1er mars (avant 23h59) par courrier électronique aux **trois** TDmen [1]. L'objet du mail devra être `[ASR2] Rendu TP04`. Les questions à rendre sont celles où un barème est indiqué. **Le rapport devra faire au maximum 2 pages.**

Les fichiers du TP sont disponibles à l'adresse suivante : `https://remy.grunblatt.org/teaching/ASR2/TP04/TP04.tgz`

Exercice 1. *Mise en route avec gdb*

Lorsqu'un programme est compilé avec l'option `-g`, le compilateur laisse dans le binaire le code source pour aider le débogueur. Il est possible à l'aide de *gdb* de voir ce code source.

1. Ouvrez le programme `hello` dans *gdb*.

On remarque que l'on se retrouve dans un interpréteur interactif, comme votre émulateur de terminal. On peut remarquer que l'invite de commande est `(gdb)`. On peut donc commencer à analyser notre programme… Mais par où commencer ? Dans *gdb*, la syntaxe pour demander de l'aide sur une commande est « `help` ». Lancer la commande sans arguments donne la liste des catégories d'aides possibles possédant chacune une myriades d'instructions. Pour vous aider dans cette jungle, il y a la documentation en ligne : `https://sourceware.org/gdb/current/onlinedocs/gdb/` pour aller plus loin. Mais dans un premier temps nous allons vous guider.
On notera que l'invite de commande gdb supporte la complétion à l'aide de la tabulation.

2. À l'aide de l'instruction `list`, affichez le code source du fichier. Que remarquez-vous ?
3. On va maintenant aller dans la fonction `int main()`, pour cela on va utiliser des *points d'arrêts* à l'aide de l'instruction… `breakpoint`, aussi abrégée en `b` puisqu'il s'agit d'une commande fondamentale lorsqu'on débogue un fichier.
   (a) Installez un point d'arrêt au niveau de la fonction `main`.
   (b) Lancez le programme à l'aide de la commande `run` et admirez gdb s'arrêter au début de la fonction `main`.
4. Maintenant qu'on est dans l'exécution du programme, on va essayer de voir ce qu'il a en réalité dans le ventre.
   (a) À l'aide de l'instruction `disassemble`, affichez le code en assembleur étendu *AT&T* du programme.
   (b) Essayez de reconnaître des bouts du code assembleur par rapport au code source.
5. Continuez l'exécution jusqu'à la ligne 7 du fichier `hello.c`, soit à l'aide d'un point d'arrêt, soit avec l'exécution pas à pas (`next`).
   (a) Essayez de changer le contenu de la variable `a` pour que le programme affiche « Bien joué ! ».
   (b) *(Optionnelle)* Relancez une seconde fois le programme, mais cette fois-ci remplacez le contenu à l'adresse mémoire où est rangée `int a` pour afficher « Bien joué ! ».

Bravo ! Vous avez résolu votre premier `crackme` !

Exercice 2. *`crackme`*

Un `crackme` est un petit programme jouet pour se familiariser avec les notions de sécurité du code.

1. `{valentin.lorentz,remy.grunblatt, alexandre.talon}@ens-lyon.fr`

1. Commencez par lancer le programme `crackme_1` une première fois pour voir ce que vous cherchez.
2. Avec *gdb* allez au niveau de `main()`.
   (a) **(/0,5)** Si vous essayez `list`, que se passe-t-il ? Pourquoi ?
   (b) On va donc devoir se reposer sur l'assembleur annoté de *gdb*. Essayez d'y reconnaître un appel de fonction (`callq`) que vous avez utilisé au TP n° 2.
   (c) Pour afficher le contenu d'un registre, d'une adresse mémoire ou autre, la commande est `print`, et il est possible de préciser le type explicite de l'argument (comme en C, où on peut caster les variables).
3. **(/2)** Quels sont les identifiant, mot de passe et phrase secrète ? (+ méthode).
4. **(/1)** Le mot de la fin : En utilisant la commande `strings` dans votre terminal, arrivez au même résultat. Décrivez comment trouver facilement les identifiants, mot de passe et texte caché (en supposant bien sûr ne pas les connaître).
5. **(/0,5)** Le mot de la fin[2] Comment trouver la phrase secrète sans chercher les identifiants (et sans utiliser `strings`) ?

Exercice 3. *Vous en reprendrez une fourchette ?*

Cette fois-ci le programme à cracker est `crackme_2`, dont on vous laisse deviner le comportement.

1. **(/1)** Essayez la méthode précédente, pourquoi ne fonctionne-t-elle pas ? Quels sont les problèmes rencontrés ?
2. **(/1)** Quel est le texte caché ? (+ méthode)
3. **(/2)** Quels est le mot de passe ? (+ méthode)

Exercice 4. *Sous le capot.*

Ce dernier exercice est plus difficile que les précédents.
Le but ici est de mieux comprendre comment `gcc` fait pour passer d'un programme C à un programme assembleur, sans aller jusqu'à un cours de compilation, le but ici est de reconnaître des motifs dans l'assembleur correspondant à des instructions en C.

1. Écrivez des programmes simples en *C* et compilez-les avec les options `-S -m32` de votre compilateur favoris. Cela va vous donner un code assembleur dans un fichier `source.s`.
   (a) Essayez de reconnaître les motifs correspondant à vos instructions **`for`**, **`if`**, **`while`**, `=`, `^=`, ...
   (b) Compilez réellement votre programme cette fois-ci (avec l'option `-m32` toujours, pour être en 32 bits, ça rend les adresses mémoires plus lisibles, c'est pour votre bien), et essayez d'en altérer le comportement avec *gdb*, sans qu'il ne provoque une erreur de segmentation.
   Vous avez désormais toutes les clefs en main pour venir à bout de `crackme_32` ! Si jamais vous n'arrivez pas à compiler avec l'option `-m32`, c'est parce que vous ne disposez pas des *headers* de fichiers correspondant à cette architecture, c'est ce qu'il se passe sur les salles machines par exemple. Dans ce cas compilez sans cette option et essayez de vous en prendre à `crackme_64`.
2. **(/1)** Quel est le texte caché ? (+ méthode)
3. **(/1)** Quel est le mot de passe ? (+ méthode)
4. **(/0,5)** Quel est le masque ? (+ méthode)

Remarque finale. Il vous reste d'autres commandes de *gdb* à découvrir pour pouvoir déboguer vos programmes, nous vous laissons aller regarder du côté de : `next`, `step`, `watch`, ...

2. bis