SEL - TP - Rapport

Introduction

Modification de code à la volée, par Antoine Geimer et Lendy Mulot.

Documentation

Nous avons implémenté les fonctions suivantes:

```
pid_t find_process(char* ownername, char* process);
int get_function_offset(char* target_process, char* target_function);
void* get_process_memory(pid_t pid);
void* get_libc_memory(pid_t pid);
int write_in_memory(pid_t pid, long address, unsigned char* buffer,
                    int len, unsigned char* override);
int run(int argc, char** argv);
```

- `find_process` renvoie le pid d'un processus étant donné le nom de l'utilisateur et le nom du processus.
- `get_function_offset`renvoie le décalage entre le début de la section texte et le code de la fonction `target_function` pour le processus `target_processus`.
- `get_process_memory` renvoie un pointeur vers le début de la section texte du processus.
- `get_libc_memory` est assez similaire à `get_process_memory`, elle renvoie un pointeur vers le début de la libc pour le processus.
- `write_in_memory` écrit la séquence d'instruction `buffer` à l'addresse `address` du processus. Les instructions écrasées sont sauvegardées dans `override` s'il n'est pas `NULL`.
- `run` est la fonction principale, appelée par le `main` (nous avons un dossier par partie et `main` appelle le `run` du fichier utilisé pour la compilation).

Avancement

Les trois premières parties ont été validées.
Voici le fonctionnement de la quatrième partie: * On récupère les arguments (nom du processus, nom de la fonction, taille du code à écrire) et on calcule les pids des processus traçant et tracé. * On calcule les adresses des fonctions `posix_memalign` et `mprotect` dans l'espace d'adressage du tracé. Pour cela on calcule l'offset de ces fonctions dans l'espace d'adressage du traçant en utilisant l'adresse (virtuelle) de la fonction et le résultat de `get_libc_memory`. L'offset étant le même pour le processus tracé, on a donc l'adresse de la fonction sur le processus tracé. * On écrit les instructions pour les appels à `posix_memalign` et `mprotect` avec gestion des registres à chaque `trap`. * On restaure les instructions écrasées et les registres. * On écrit les instructions pour le `jmp` vers la fonction écrite sur le tas.

Améliorations possibles

- Notre implémentation ne peut actuellement écrire qu'une seule fonction particulière, qui est écrit directement dans le fichier `part4/part.c` (qui correspond au code de la fonction `optimised` du fichier `target.c`). Il pourrait être intéressant de remplacer le paramètre de taille par un chemin vers un fichier contenant les instructions à écrire.
- On ne peut actuellement pas remplacer la fonction optimisée par une fonction faisant des appels à d'autres fonctions. En effet, si les `call` sont à des adresses relatives, le code ne sera plus correcte lorsqu'on le met dans le tas. On pourrait éventuellement remplacer toutes les instructions `call` relatives par des `call` absolus (il existe peut-être une option de compilation permettant cela).

Challenge bonus

TODO

Comment utiliser

- Cloner le répo: `git clone git@github.com:ZeGmX/SEL-TP.git`.
- Lancer un terminal dans le dossier `SEL-TP`.
- Pour compiler la partie `i, i = 1, 2, 3 ou 4`, utiliser la commande `make PART=i`.
- Lancer le processus tracé avec `./target`.
- Lancer le processus traçant avec `./tp` suivi des arguments nécessaire pour la partie. Utiliser la commande `./tp` affichera un message d'erreur indiquand quels sont les arguments attendus pour la partie. Pour la partie 4, vous pouvez utiliser `./tp target target_function 11`.
- Si vous utilisez le processus `target`, entrez une valeur entière dans son terminal pour que le tracé puisse reprendre (il attend sur un `scanf`) et que le traçant fasse son travail.