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

- Dans les différentes parties, nous avons montré comment passer jusqu'à trois arguments en paramètres de fonction, utilisant les registres, cependant pour un plus grand nombre d'arguments, le fonctionnement est différent et n'as pas été implémenté.
- 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).
- Même si on résolvait le problème précédent, cela ne résoudrait pas des problèmes tels qu'un appel de fonction en utilisant une variable statique non définie dans le tracé (par exemple une chaîne de caractère pour `printf`). Il faudrait ici allouer de la place pour placer la chaîne et remplacer son adresse initiale relative (obtenue en la compilant) par une adresse absolue.

Challenge bonus

Remplacement des appels

Pour remplacer les appels à la fonction non-optimisée par les appels à la fonction optimisée, deux options sont possibles:

- On implémente une sorte de `parser` d'instructions x86 pour identifier les `call` et les remplacer par des `call` absolus à l'adresse de la fonction optimisée (en ajoutant éventuellement un instruction pour mettre l'adresse dans un registre si besoin) ou en recalculant l'*offset*.
- On utilise la sortie de `objdump` pour identifier plus facilement les `call` puis on applique la même idée que pour la première option.

Cependant, devoir ajouter une instruction pour mettre l'adresse dans un registre demanderait de décaler les instructions suivantes, pouvant déborder sur une zone non allouée. Recalculer l'*offset* serait donc probablement l'option à privilégier.

Multi-*threads*

Pour généraliser cette injection de code, on s'intéresse au cas multi-*thread* .
Ce qui ne change pas :

- Le principe d'appel de fonction, on place les paramètres dans les registres, on fait un `call`, etc.
- Les *threads* partageant le même tas, l'allocation de mémoire pour écrire la fonction optimisée fonctionnera sur le même principe.
- Le remplacement des appels (ou la méthode trampoline) seront identiques puisque le code n'est pas dupliqué sur les *threads*.

Les nouvelles difficultés :

- On doit s'assurer qu'un seul des *threads* fasse un appel à `posix_memalign`. Même si plusieurs appels ne sont pas intrinsèquement gênants, il serait souhaitable de ne pas allouer de mémoire inutilement.
- On doit relancer chaque *thread*, et sauvegarder/restaurer les registres pour chaque *thread*.
- Pour les appels de fonctions, il faut faire attention à modifier les registres du bon *thread* (celui qui va passer par l'instruction `call`) puisque chaque *thread* possède son propre jeu de registres.

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`, attendez quelques instants (`target` contient une instruction `sleep(5)`).

Une execution correcte de la partie 4 devrait ressembler à ceci:

```
CHALLENGE 4
Found process ID : 4744
Found target function address 55810615e155
Found posix_memalign address: 7f27772a9d90
Found mprotect address: 7f2777318200
Wrote 13 byte(s) into memory.
Target got a signal : Trace/breakpoint trap
Allocated address: 558107ee8000
Wrote 13 byte(s) into memory.
Target process ran mprotect successfully.
Wrote 11 byte(s) into memory.
Wrote 2 byte(s) into memory.
Wrote 8 byte(s) into memory.
Wrote 2 byte(s) into memory.
```

Le processus tracé quant à lui devrait avoir un comportement similaire au suivant (les adresses peuvent différer):

```
This program will run target_function every 5 seconds.
Running main loop...
Entering original target_function!
Return value: 5
Running main loop...
Entering original target_function!
Return value: 5
Running main loop...   <- code injection happened here
Return value: 1
Running main loop...
Return value: 1
Running main loop...
Return value: 1
```