My ltrace

Version 1
9 Décembre 2013

Epita systems/security laboratory 2014 <recrutement@lse.epita.fr>

Copyright

▷ Assurez vous d'avoir la dernière version de ce document.

Table des matières

Obligations

- ▷ Lire le sujet en *entier*
- ▷ Respecter les règles
- ▷ Respecter l'heure de rendu

Rendu

Responsables du projet	Ivan DELALANDE < colona@lse.epita.fr> Franck MICHEA < kushou@lse.epita.fr> Clément ROUAULT < hakril@lse.epita.fr>
Balise du projet :	**[LTRC]**
Développeurs par équipe:	1
Procédure de rendu :	Git ACU
Rendu :	22/12/2012 at 11h42
Durée du projet :	2 weeks
Groupe de discussion :	iit.labos.lse.recrutement
Architecture/OS :	Linux x86_64
Langage(s) :	C++
Compilateur :	**g++ | clang**
Options du compilateur :	**-Wall -Wextra** (au minimum, C++11 est autorisé, de même pour boost)
Includes autorisés :	All

Consignes

Les informations suivantes sont très importantes :

Le non-respect d'une des consignes suivantes entraînera des sanctions pouvant aller jusqu'à la multiplication de la note finale par 0.

Ces consignes sont claires, non-ambiguës, et ont un objectif précis. En outre, elles ne sont pas négociables.

N'hésitez pas à demander si vous ne comprenez pas une des règles.

- **Consigne 0:** Le fichier `AUTHORS` doit être présent à la racine de votre rendu. Il comprend une étoile **`*`**, une espace, votre login (ex : login_x) et un retour à la ligne.

  Exemple :

  ```
  42sh$ cat -e AUTHORS
  * login_x$
  42sh$
  ```

  L'absence du fichier `AUTHORS` entrainera un 0 non négociable.
- **Consigne 1:** Lorsque des formats de sortie vous sont donnés en exemple, vous devez les respecter scrupuleusement. Les exemples sont indentés : **`42sh$`** représente le prompt, utilisez le comme repère.
- **Consigne 2:** Le respect de la "coding style" est obligatoire si elle existe dans le langage demandé. Dans tous les cas, le code rendu doit être propre, lisible et ne doit pas dépasser 80 colonnes.
- **Consigne 3:** Les droits sur les fichiers et les dossiers sont obligatoires pour tout le projet : répertoire principal, fichiers, sous répertoires ...
- **Consigne 4:** Si un exécutable est demandé, vous devez uniquement fournir ses sources dans le répertoire src, sauf mention contraire. Ils seront compilés par nos soins.
- **Consigne 5:** Votre rendu doit respecter l'aborescence suivante, où **`login_x`** doit être remplacé par votre login ou celui de votre chef de groupe:

  ```
  login_x-my_ltrace/AUTHORS *
  login_x-my_ltrace/README *
  login_x-my_ltrace/TODO *
  login_x-my_ltrace/src/ *
  ```

  Les fichiers suivants sont requis :

`TODO`	décrit les tâches à accomplir. Il doit être régulièrement mis à jour.
`AUTHORS`	contient les auteurs du projet. Doit respecter la norme EPITA (spécifiée plus haut).
`README`	décrit le projet et les algorithmes utilisés dans un anglais correct. Explique aussi comment utiliser votre projet.

- **Consigne 6:** Une archive non propre est une archive qui contient des fichiers interdits (**`*~`**, **`*.o`**, **`*.a`**, **`*.so`**, **`*#*`**, **`*core`**, **`*.log`**, binaires, etc.).

  Une archive non propre est aussi une archive dont le contenu n'a pas les bons droits.

  Une archive non propre entraîne automatiquement la perte de deux points sur votre travail.
- **Consigne 7:** Vous devez rendre à l'heure. Tout retard, même d'une seconde, entraînera au mieux la note de 0 non négociable.

- ▷ **Consigne 8:** Toutes les fonctions et commandes qui ne sont pas explicitement autorisées sont interdites. Les abus peuvent entraîner jusqu'à l'obtention de la note, non négociable, de -21.
- ▷ **Consigne 9:** La triche, l'échange de code source, de tests, d'outils de tests et de correction de norme, sont pénalisés par une note, non négociable, de -42.
- ▷ **Conseil:** N'attendez pas la dernière minute pour commencer le projet.

Introduction

Ce projet est l'occasion de montrer que vous êtes aptes à travailler sur des projets internes du LSE. Vous allez aborder de multiples notions qui ont toutes trait à la manipulation de binaires produits par votre compilateur, aussi bien de manière statique que dynamique.

Vous allez découvrir différentes manières d'étudier un programme en exécution et éventuellement d'agir sur son exécution.

Si vous avez la moindre question, n'hésitez pas à nous demander sur les news, sur le channel IRC **`#lse-recrut@irc.epita.fr`** ou, pour des demandes exceptionnelles, par email sur **`recrutement@lse.epita.fr`**.

Amusez vous bien et impressionnez nous !

1 Level 1: my_strace

Nom du binaire de rendu :	**my_strace**
Répertoire de rendu :	**login_x-my_ltrace/src/my_strace/**
Droits :	640 pour le fichier, 750 pour le répertoire
Includes recommandés :	sys/ptrace.h, sys/user.h

Synopsis

```
./my_strace /path/to/traced/binary [args]...
```

1.1 Objectif

Dans cet exercice, on attend de vous que vous produisiez un programme similaire à **`strace(1)`** (testez le !), c'est à dire une solution capable d'afficher la liste des syscalls effectués par un autre processus.

1.2 Principe

Vous allez devoir, pour cela, lancer le binaire passé en argument à votre **`strace(1)`**, en spécifiant qu'il doit être tracé par son père (regardez la commande **`PTRACE_TRACEME`** de **`ptrace(2)`**). Vous vous servirez alors de la commande **`PTRACE_SYSCALL`** de **`ptrace(2)`** pour redonner la main au père au début et à la fin de chaque syscall, ce qui vous donnera l'occasion d'analyser le contenu des registres et de la mémoire afin de déterminer quel syscall est en train d'être appelé.

1.3 Convention d'appel des syscalls

Dans le cas d'un appel système, tous ses arguments sont passé par registres sur **`x86`** et **`x86_64`**. Pour vous aider à comprendre comment il se déroule, voici les différentes étapes d'un appel système sur **`x86_64`** :

- placer le numéro de l'appel système dans le registre **`rax`**,
- placer les arguments de l'appel système, de gauche à droite, dans les registres suivants : **`rdi`**, **`rsi`**, **`rdx`**, **`r10`**, **`r8`**, **`r9`**,
- effectuer une instructiond d'appel système qui passe la main au kernel (**`int 0x80`**, **`sysenter`**, **`syscall`**... selon les plateformes),
- le kernel repères les informations qui lui sont utiles, effectue le travail et retourne,
- récupérer la valeur de retour de l'appel système dans le registre **`rax`**.

Connaissant cette convention d'appel, vous pouvez récupérer les informations qui vous intéressent en analysant les registres, avant et après l'exécution de l'appel système.

1.4 Palier 1 : affichage des syscalls

Le premier palier consiste à afficher simplement la liste des appels systèmes que fait un binaire, suivi de leur valeur de retour (au format décimal). Pour avoir une correspondance entre un numéro de syscall et son nom, regardez le fichier **`/usr/include/asm/unistd_64.h`**.

De plus, vous devez afficher un message indiquant le code de retour du binaire tracé. Voyez l'exemple fourni ci-dessous pour le format.

Le format de sortie est simple, le nom de l'appel système, suivi de parenthèses ouvrantes et fermantes, et enfin, la valeur de retour séparée par un signe égal. Vous afficherez un appel système par ligne. La liste des appels système sera affichée sur **`STDERR`**.

Par exemple :

```
42sh$ ./my_strace /bin/ls
brk() = 6557696
mmap() = 140077684056064
access() = -2
open() = 3
...
close() = 0
munmap() = 0
close() = 0
program exited with code 0
42sh$
```

1.5 Palier 2 : affichage des arguments

Après avoir affiché la liste des appels système, une fonctionnalité qui peut se révéler assez plaisante est l'affichage des arguments des ces appels.

Il vous suffira d'analyser l'état des registres au début de l'appel système pour déterminer les arguments passés à une commande.

Étant donné qu'il est assez fastidieux de traiter tous les appels systèmes, nous vous conseillons de ne gérer que les plus importants ou les plus utilisés. Un bon point de départ est de gérer les classiques **`open(2)`**, **`close(2)`**, **`read(2)`**, **`write(2)`**, **`mmap(2)`**, **`munmap(2)`** et **`fstat(2)`**.

Le format de sortie devra être homogène avec celui du palier 1, avec les arguments affichés entre les parenthèses et nommés comme dans le **`man(1)`** correspondant.

Par exemple :

```
42sh$ ./my_strace /bin/ls
brk() = 6557696
mmap(addr = 0x0, length = 4096, prot = 3, flags = -1, fd = -1, offset = 0) =
140077684056064
access() = -2
open(pathame = "/etc/ld.so.cache", flags = 1) = 3
...
close(fd = 1) = 0
munmap(addr = 0x7f6660995000, length = 4096) = 0
close(fd = 2) = 0
program exited with code 0
42sh$
```

Ce palier sera évalué en soutenance. Il n'est donc pas indispensable de gérer tous les syscalls, mais une gestion extensive de ceux-ci vous rapportera quelques points en bonus. ;)

1.6 Bonus : Filtrage des syscalls

La dernière étape de votre **`my_strace`** sera d'apporter une petite touche de sécurité au tout. Vous allez implémenter des fonctionnalités similaires à celles offertes par la commande **`systrace(1)`**. Vous pourrez alors avoir une forme spéciale de sandboxing pour tester le comportement de certains programmes. Par exemple, que se passe-t-il si vous lancez le programme **`ls(1)`** en interdisant tous les **`open(2)`** ?

Vous allez dans un premier temps tenter de modifier la valeur de retour d'un appel système. Pour rappel, il s'agit simplement de modifier le contenu d'un registre en sortie de syscall.

La deuxième étape consiste à interdire un syscall. En entrée d'appel système, vous regardez si le numéro présent dans **`rax`** est autorisé. Si oui, l'appel se déroule comme prévu, sinon, vous empêchez l'exécution et vous retournez immédiatement -1.

Ce bonus sera évalué en soutenance, vous êtes donc libre sur le format d'entrée des règles de filtrage. Vous pouvez les passer par la ligne de commande, ou par l'intermédiaire d'un fichier de configuration.

2 Level 2 : my_ltrace

Nom du binaire de rendu :	**my_ltrace**
Répertoire de rendu :	**login_x-my_ltrace/src/my_ltrace/**
Droits :	640 pour le fichier, 750 pour le répertoire
Includes recommandés :	sys/ptrace.h, sys/user.h

Synopsis

```
./my_ltrace /path/to/ltraced/binary [args]...
```

2.1 Objectif

Le but de cet exercice est de fournir un binaire similaire a **`ltrace(1)`**.

C'est à dire une solution capable d'afficher l'ensemble des appels fait à différentes bibliothèques dynamiques.

2.2 Principe

Pour cela vous allez devoir lancer le programme donné en paramètre à **`ltrace`**. Vous devrez être capable de lister l'ensemble des appels fait à des bibliothèques dynamiques. Vous devrez exploiter les structures mises en place par le format ELF tel que la PLT et la GOT. Les différents niveaux de cet exercice utilisent diverses façons d'arriver à cette fin.

2.3 Palier 1 : Learn to PLT/GOT

Le premier palier consiste à afficher l'ensemble des fonctions externes pouvant être appelées par le binaire. Pour cela vous allez devoir explorer la GOT à travers les *relocations* de celui-ci et en extraire les symboles associés.

2.4 Palier 2 : Breakpoints

Maintenant que vous savez trouver et interpréter les entrées de la GOT de votre programme, il ne vous reste plus qu'à mettre en place des breakpoints pour savoir quand une certaine fonction est appelée.

Pour simplifier le travail vous voudrez peut être désactiver le *lazy-biding* du linker dynamique (**`ld.so(8)`** : **`LD_BIND_NOW`**).

Les principe est donc :

— un breakpoint (ou instruction **`int 3`**) correspond à l'opcode **`0xCC`**. Il vous suffit donc d'écrire cette valeur en mémoire pour placer un breakpoint !

— lorsque le processeur tombe sur une telle instruction il passe la main au programme qui debug. À ce moment là vous n'avez qu'à vérifier quel breakpoint a été trigger pour connaître la fonction appelée.

Conseil : lorsque vous tomberez sur un breakpoint, n'oubliez pas de restaurer l'ancienne valeur pour exécuter l'instruction correcte et de remettre le breakpoint ensuite (voir **`PTRACE_SINGLESTEP`**).

2.5 Palier 3: Hooks

Vous avez maintenant un ltrace parfaitement fonctionnel (bravo !). Cependant, un problème demeure, ltrace est constamment attaché au processus cible. Vous devez maintenant faire en sorte que votre ltrace puisse se détacher du processus cible tout en ayant un tracing fonctionnel.

Pour cela votre **ltrace** va devoir poser un breakpoint sur **main**, injecter du code dans le processus et se détacher. Dans un premier temps, pour empêcher le linker dynamique de réécrire la GOT vous pouvez utiliser (**LD_BIND_NOT**).

Principe :

— dans un premier temps, vous allez devoir injecter du code dans le processus cible,
— ce code permettra par exemple d'afficher une string sur la sortie d'erreur,
— ensuite vous devrez hook la GOT : c'est à dire changer les valeurs dans la GOT pour sauter vers du code à vous qui :
  — affichera l'appel avec le nom de la fonction et les valeurs de ses paramètres,
  — lancera l'appel à la bonne fonction,
  — affichera la valeur du résultat.

Vous pouvez gérer comme vous voulez la gestion des changements dans la GOT (**LD_BIND_NOT** ou **LD_BIND_NOW** étant les plus simples).

2.6 Bonus

2.6.1 Afficher le pointeur d'instruction (facile)

Ajoutez l'option **-i** qui permettra d'afficher l'adresse de l'instruction déclenchant l'appel tracé.

2.6.2 Afficher la backtrace (-fno-omit-frame-pointer) (facile)

À chaque appel que vous tracez, affichez la backtrace de cette appel.

2.6.3 Attacher à un processus arbitraire (facile)

Ajoutez l'option **-p PID** qui permettra d'indiquer à **ltrace** de s'attacher à un processus déjà lancé, en prenant en argument le numéro de **PID** du processus.

2.6.4 Tracer les fils (moyen)

Ajoutez l'option **-f** qui permettra d'indiquer à **ltrace** de s'attacher également à tous les fils créés par un programme, par le biais des appels systèmes **fork(2)** ou **clone(2)**.

2.6.5 Afficher les signaux (moyen)

Affichez les signaux reçu par le processus tracé.

2.6.6 Filtrer les appels (moyen)

Ajoutez l'option **-e FILTER** qui permettra de filtrer les fonctions tracées selon une syntaxe décrite dans la man-page de **ltrace(1)**. Vous n'êtes bien sûr pas obligé de tout supporter

2.6.7 Attacher à un processus arbitraire (LD_BIND_LAZY) (difficile)

Ajoutez l'option **-p PID** qui permettra d'indiquer à **ltrace** de s'attacher à un processus déjà lancé, en prenant en argument le numéro de **PID** du processus.

2.6.8 Afficher la backtrace (difficile)

À chaque appel que vous tracez, affichez la backtrace de cette appel.

2.6.9 Gérer les binaires PE (difficile)

Bon courage !

2.6.10 Free-porn

Impressione nous !

3 CTF

N'oubliez pas le CTF[1] !;)

1. http://ctf.lse.epita.fr/