My Debugging Suite

Vive ptrace !

Version 1
06 décembre 2010

Laboratoire Sécurité/Système d'EPITA 2012 <contact@lse.epita.fr>

Copyright

Table des matières

Obligations

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

Rendu

Responsables du projet	Gabriel Laskar < gabriel@lse.epita.fr> Stephane Sezer < stephane@lse.epita.fr>
Balise du projet :	**[DBS]**
Développeurs par équipe:	1
Procédure de rendu :	Upload sur l'intranet
Méthode de rendu:	login_x-mydbs.tar.bz2
Rendu :	17/12/2010 à 23h42
Durée du projet :	2 semaines
Groupe de discussion :	iit.labos.lse.recrutement
Architecture/OS :	i386 Fedora
Langage(s) :	C++
Compilateur :	**g++**
Options du compilateur :	**-Wall -Werror -Wextra**
Includes autorisés :	Tous

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-mydbs/AUTHORS *`
  `login_x-mydbs/README *`
  `login_x-mydbs/TODO *`
  `login_x-mydbs/src/ *`
  `login_x-mydbs/src/my_nm/ *`
  `login_x-mydbs/src/my_nm/Makefile *`
  `login_x-mydbs/src/my_nm/my_nm.cc`
  `login_x-mydbs/src/my_strace/ *`
  `login_x-mydbs/src/my_strace/Makefile *`
  `login_x-mydbs/src/my_strace/my_strace.cc`
  `login_x-mydbs/src/my_db/ *`
  `login_x-mydbs/src/my_db/Makefile *`
  `login_x-mydbs/src/my_db/my_db.cc`
  `login_x-mydbs/src/my_prof/ *`
  `login_x-mydbs/src/my_prof/Makefile *`
  `login_x-mydbs/src/my_prof/my_prof.cc`

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:** Pour tout problème relatif au projet, vous pouvez entrer en contact avec les assistants en envoyant un ticket à partir de l'intranet dans l'onglet **`Ticket`** du projet.
- ▷ **Consigne 8:** Supprimer le répertoire **`~/../rendu`** est interdit durant toute l'année.
- ▷ **Consigne 9:** Vous devez rendre à l'heure. Tout retard, même d'une seconde, entraînera au mieux la note de 0 non négociable.
- ▷ **Consigne 10:** 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 11:** 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.

1 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 voir dans un premier temps comment récupérer des informations sur un fichier binaire en étudiant la structure du format ELF puis analyser les informations présentes à l'exécution d'un programme en analysant sa mémoire, son flot d'instruction, les appels de fonction, etc.

Amusez-vous bien !

2 my_nm

Nom du binaire de rendu :	**my_nm**
Répertoire de rendu :	**login_x-mydbs/src/my_nm/**
Droits :	640 pour les fichier, 750 pour le répertoire
Includes recommandés :	elf.h, sys/mman.h

Synopsis

```
./my_nm /path/to/binary
```

2.1 Objectif

Dans cet exercice, on attend de vous que vous produisiez un programme similaire à **nm(1)**, c'est à dire une solution capable d'afficher la liste des symboles d'un fichier ELF.

2.2 Principe

Le principe est simple, vous ouvrez le binaire passé en paramètre, vérifiez que c'est un ELF, et ensuite aller cherchez la section contenant les symboles (**.symtab**) pour l'afficher. Il vous faudra aussi lire certaines autres sections de manière a pouvoir récupérer les chaines de caractères correspondantes.

Vous n'avez pas à afficher les symboles des fichiers, mais tous les autres sont nécessaires.

2.3 Champs a afficher

Votre binaire se comporte comme **readelf(1)** plutôt que comme **nm(1)**. Voici les 2 sorties sur un binaire simple :

```
42sh$ cat example/example.c
#include <stdio.h>

int do_calc(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 12;
    int b = 30;

    printf("affichage de %i + %i\n", a, b);

    printf("= %i\n", do_calc(a, b));

    return 0;
}
42sh$ gcc -c example/example.c
42sh$ readelf -s example.o

Symbol table '.symtab' contains 11 entries:
```

```
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS example.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000    14 FUNC    GLOBAL DEFAULT    1 do_calc
     9: 0000000e    98 FUNC    GLOBAL DEFAULT    1 main
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
42sh$ ./my_nm example.o
00000000 0      STT_NOTYPE      STB_LOCAL       STV_DEFAULT     UND
00000000 0      STT_SECTION     STB_LOCAL       STV_DEFAULT     .text
00000000 0      STT_SECTION     STB_LOCAL       STV_DEFAULT     .data
00000000 0      STT_SECTION     STB_LOCAL       STV_DEFAULT     .bss
00000000 0      STT_SECTION     STB_LOCAL       STV_DEFAULT     .rodata
00000000 0      STT_SECTION     STB_LOCAL       STV_DEFAULT     .note.GNU-stack
00000000 0      STT_SECTION     STB_LOCAL       STV_DEFAULT     .comment
00000000 14     STT_FUNC        STB_GLOBAL      STV_DEFAULT     .text   do_calc
0000000e 98     STT_FUNC        STB_GLOBAL      STV_DEFAULT     .text   main
00000000 0      STT_NOTYPE      STB_GLOBAL      STV_DEFAULT     UND     printf
42sh$
```

2.4 Conseils pour cet exercice

- Cet exercice est très simple, ne vous embêtez pas trop avec les sorties.
- Il y a des tabulations entre chaque champ.
- Arrangez vous pour pouvoir réutiliser le code qui récupère la liste des symboles, vous en aurez besoin plus tard.
- Vous êtes en C++, pensez itérateurs.
- Tout est fourni dans le manuel de l'elf (**`elf(5)`**).
- Vérifiez bien si votre fichier est un ELF valide.
- Votre code doit marcher avec n'importe quel fichier ELF, pas que des objets.

3 my_strace

Nom du binaire de rendu :	**my_strace**
Répertoire de rendu :	**login_x-mydbs/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]...
```

3.1 Objectif

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

3.2 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.

3.3 Convention d'appel des syscalls

Un appel système ne se produit pas de la même manière qu'un appel de fonction standard. Les arguments ne sont pas passés par la pile mais par registres. Nous vous fournissons, pour vous aider, la liste des étapes d'un appel système.

- Placer le numéro de l'appel système dans le registre eax
- Placer les arguments de l'appel système, de gauche à droite, dans les registres suivants : ebx, ecx, edx, esi, edi, ebp
- Effectuer une interruption 0x80 qui passe la main au kernel
- ... 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 eax

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.

3.4 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_32.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$
```

3.5 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 ;)

3.6 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 eax est autorisé. Si oui, l'appel se déroule comme prévu, si non, 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.

4 my_db

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

Synopsis

```
./my_db /path/to/binary [args]...
```

4.1 Objectif

Maintenant que vous vous êtes bien familiarisés avec **`ptrace(2)`**, on va vous demander d'écrire un petit debugger.

4.2 Découpage

Cet exercice est découpé en 5 parties, la seule partie obligatoire est la première. Pour le reste, vous faites ce que vous voulez.

4.3 Palier 1 : Commandes de base

Notre debugger est un simple interpréteur de commandes rudimentaires, vous allez donc lire des commandes depuis l'entrée standard et écrire le résultat sur la sortie standard.

Le programme debugge sera lance des le debut avec un **`PTRACE_TRACEME`** juste avant.

Voici la liste des commandes de base à implémenter :

- **`quit`** : quitte le debugger.
- **`kill`** : tue le processus en cours de debug.
- **`continue`** : continue l'exécution du processus
- **`registers`** : affiche la liste des registres du processus en cours de debug.

Ces commandes sont très basiques, voici tout de même un exemple de sortie de **`continue`** et **`register`**. Notez comment arrêter l'exécution d'un processus avec **`int3`**.

```
42sh$ cat tests/break.c
#include <stdio.h>

int main()
{
  puts("    before breakpoint");

  asm volatile ("int3\n");

  puts("    after breakpoint");

  return 0;
}
42sh$ gcc tests/break.c
42sh$ ./my_db ./a.out
continue
    before breakpoint
registers
```

```
registers
eax 22
ebx 3078311924
ecx 3078316864
edx 22
esp 0xbf83c9d0
ebp 0xbf83c9e8
esi 0
edi 0
eip 0x80483ba
eflags 582
cs 115
ss 123
ds 123
es 123
fs 0
gs 51
orig_eax 4294967295

continue
    after breakpoint
kill
quit
42sh$
```

Interlude

Bon, OK, personne ne vous a expliqué ce qu'est **int3** et les mots-clés bizarres dans le code C de l'exemple d'au dessus.

int3 est une instruction assembleur qui permet d'envoyer l'interruption numéro 3 au processeur. C'est l'interruption qui correspond à un breakpoint. Oui, c'est un breakpoint, rien de plus. Le programme s'interrompt, et nous donne la main au père qui est en train d'attendre avec un **wait(2)**.

4.4 Palier 2 : Memdump

Vous allez maintenant essayer d'afficher le contenu de la mémoire du processus tracé.
Exemple :

```
42sh$ ./my_db /path/to/traced/binary [args]...
continue
registers
x 10 $eip
continue
quit
```

Vous avez 3 commandes à implémenter. Le format général de ces commandes est :

```
command count addr
```

- **command** vaut **x**, **d** ou **u** pour afficher, respectivement, en hexadécimal, décimal signé ou décimal non signé.
- **count** est le nombre de valeurs de 32 bits que vous devez afficher.
- **addr** est l'adresse (en décimal ou héxadecimal) à partir de laquelle afficher. Elle peut aussi être un symbole du binaire.

4.5 Palier 3 : Exécution pas à pas

On va maintenant exécuter du code pas à pas. Pour cela vous devez implémenter une commande **`next`** qui prendra 0 ou 1 argument qui sera le nombre de pas à faire.

```
42sh$ ./my_db /path/to/binary [args]...
continue
registers
next
registers
next 3
registers
continue
quit
```

4.6 Palier 4 : Breakpoints

Vu que vous commencez à bien maitriser **`ptrace(2)`**, vous allez pouvoir attaquer les choses sérieuses : poser des points d'arrêt sur votre programme.

Vous aurez 3 commandes à implémenter :

- **`break`** : suivie d'un argument, cette commande place un breakpoint à l'adresse donnée. L'argument doit être une adresse (en hexadécimal ou décimal) ou un symbole contenu dans l'ELF (attention à ce qu'il soit du bon type, il serait plutôt dommage de placer un breakpoint sur une variable globale par exemple).
- **`blist`** : affiche la liste des breakpoints placés dans le code.
- **`bdel`** : supprime le breakpoint dont le numéro est affiché dans la liste.

Voici un exemple de sortie :

```
42sh$ cat tests/break.c
#include <stdio.h>

void func1(void)
{
  printf("test1\n");
}

void func2(void)
{
  printf("test2\n");
}

int main()
{
  func1();
  func2();
}
42sh$ gcc tests/break.c
42sh$ ./my_db ./a.out
b func1
b func2
blist
1 0x080483a4 func1
2 0x080483b8 func2
```

```
bdel 1
blist
1 0x080483b8 func2
quit
42sh$
```

Principe

Pendant tout le début du projet, vous avez simplement mis des **int3** dans votre code pour arrêter l'exécution. Vous allez maintenant passer à l'étape supérieure et poser des breakpoints directement à partir de votre debugger.

L'instruction **int3** correspond à l'opcode **0xCC**. Il vous suffit donc d'écrire cet opcode dans votre code chargé en mémoire, à l'emplacement voulu pour avoir un breakpoint.

Simple non ? Bon, il faut de plus que l'on puisse passer par dessus pour pouvoir continuer l'exécution normalement, en exécutant l'instruction que l'on a remplacé par le **int3**. L'eip au moment du break sera placé juste après. Il faut donc replacer l'ancienne valeur, mettre l'eip à la bonne adresse, faire un **next** et replacer le breakpoint.

Faites attention tout de meme, pour éviter de casser votre affichage de la callstack, reflechissez bien à l'endroit où vous placez vos breakpoints dans le cas d'un breakpoint sur une fonction. Par exemple, il peut être judicieux de placer votre breakpoint après le prologue.

4.7 Bonus : Callstack

On va maintenant afficher la liste des fonctions appelées à un moment donné (comme **bt** dans **gdb(1)**).

La commande à implémenter s'appellera **backtrace** et devra afficher la liste des fonctions visibles dans la pile.

5 my_prof

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

Synopsis

```
./my_prof [ -o dump.out ] [ -d dump.dot ] /path/to/traced/binary
    [args]...
```

5.1 Objectif

Le but de cet exercice est de produire un outil de profiling de code qui sera capable de déterminer les fonctions appelées, combien de fois elles sont appelées, dans quel ordre, ainsi que d'établir un graphe d'appel relatif à une exécution.

Les fichiers de sortie sont passés au programme par les options -o et -b. Si une de ces options est manquante, le fichier de sortie correspondant ne sera pas traité.

5.2 Principe

Vous allez commencer par utiliser le code de votre **`my_nm`** pour récupérer la liste de toutes les fonctions de votre binaire. Bien entendu, nous n'analysons pas les fonctions non définies de votre binaire, mais uniquement celles qui sont effectivement présentes. Après cela, il vous suffit de placer un breakpoint sur chacune des fonctions, chose que vous êtes capables de faire si vous avez fait **`my_db`**. A chaque break, vous pourrez alors mettre à jour votre compteur d'appels.

La production du graphe d'appel ne sera pas fondamentalement plus compliqué, il vous suffira d'analyser la stack pour déterminer la fonction appelante et envoyer l'information dans le ficher dot de sortie.

5.3 Palier 1 : Compteur de fonctions

Le premier palier consiste simplement à afficher la liste des appels de fonction du binaire et le nombre de fois ou elles ont été appelées.

Encore une fois, le format de sortie est particulièrement simple. Il faudra juste afficher le nom de la fonction, suivi d'une espace et du nombre d'appels, le tout dans le fichier spécifié par l'option -o du binaire.

Par exemple :

```
42sh$ ./my_prof -o dump.out ./test
42sh$ cat dump.out
func1 2
fonction_test 72
func72 0
kikoo_func 32
42sh$
```

5.4 Palier 2 : Call graph

Une fois que vous êtes capable de break au début de chaque fonction et de compter le nombre d'appel de chacune, vous pouvez produire un graphe des appels.

Le format de sortie est un fichier dot. Chaque fonction sera représenté par un sommet du graphe, et chaque appel de fonction par un arc liant les deux sommets correspondants. Une fonction func1 appelant une fonction func2 sera explicitée dans le graphe par un arc allant de func1 vers func2. Bien entendu, la présence de cycles dans votre graphe ne sera pas un problème.

Ce palier sera évalué en soutenance, vous pouvez donc vous donner le plus de mal possible pour avoir un joli graphe d'appels ;)

5.5 Bonus : Analyse des points chauds

En bonus à votre profiler, nous serions très heureux d'avoir des informations de timing pour chaque fonction. Vous afficherez les informations de timing inclusif et exclusif.

Vous avez plusieurs moyens d'implémenter cette fonctionnalité, les fonctions de gestion du temps standard, ou alors les compteurs de performance des processeurs x86 pour ne citer qu'eux.

N'attaquez ce bonus que si vous avez fini tout le reste...