liaison dynamique sous Linux

Frédéric Ferrandis (frederic.ferrandis@openwide.fr)

Juin 2010

MODIFICATIONS

VERSION	***DATE***	***AUTEUR(S)***	***DESCRIPTION***
1.0	05/07/10	F.Ferrandis	Création

Table des matières

1.Présentation du document

L'objectif de ce document est de présenter (succintement) les mécanismes permettant de charger les librairies nécessaires, à l'exécution d'un programme.

Pour cela, sera abordé le programme ld-linux.so, ainsi que les diverses variables d'environnements permettant de modifier son comportement.

Enfin, l'écriture de ce programme a conduit à la création d'une librairie utilisable depuis n'importe quelle application. Son utilisation peut permettre de créer des mécanismes de plugins ou de détournement de fonction. L'API de cette librairie sera également abordée.

2.Présentation de ld-linux.so

Le programme ld-linux.so est le programme permettant effectuant la liaison dynamique d'un programme avec des librairies dynamiques, ou bien d'une librairie dynamique avec d'autres librairies dynamiques (dont elle dépend).

Soit le programme (A) nécessitant la librairie libc.so et libdl.so pour pouvoir s'exécuter, alors le programme ld-linux.so a pour tache de charger ces deux librairies en mémoire.

Soit la librairie libutil.so, utilisant des fonctions de la libdl.so, et compilée avec l'option '-ldl' (permettant d'avoir d'avoir des références sur les symboles externes, plutôt que d'ajouter cette option à la compilation du programme utilisant la librairie), alors le programme ld-linux.so a pour tache de charger libdl.so.

2.1.Fonctionnement simplifié de ld-linux.so

Lorsqu'un programme est exécuté, une des premières actions devant être faite est le chargement des librairies dynamiques nécessaires au bon fonctionnement du programme. Pour cela, il existe un certain nombre d'information dans le binaire (qui, par défaut, est au format ELF sous Linux), permettant de savoir :

- quelles librairies dynamiques sont nécessaires pour l'exécution du programme (par défaut, il y'a au minimum la libc)
- quel programme utiliser pour faire cette recherche de librairie, et pour les charger.

Pour connaître le nom de l'application qui va devoir charger les librairies dynamiques utilisées par un programme, il existe un segment particulier dans votre programme, créé à la compilation.

Ce segment, généralement appelé segment « INTERP », contient, sous forme de chaîne de caractères, le chemin d'accés vers le programme devant charger en mémoire les librairies dynamiques d'un programme.

Généralement, ce segment n'est présent que dans les fichiers binaires exécutables, car une projection mémoire de librairie dynamique est la conséquence de l'exécution d'un programme, mais peut être également présent dans un fichier de type librairie dynamique.

Pour vérifier que le programme utilisé sera bien ld-linux.so, vous pouvez utiliser la commande suivante :

```
fred@kdiv ~ >readelf -l ./mtest

Elf file type is EXEC (Executable file)
Entry point 0x80483b0
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0062c 0x0062c R E 0x1000
  LOAD           0x00062c 0x0804962c 0x0804962c 0x00114 0x0011c RW  0x1000
  DYNAMIC        0x000640 0x08049640 0x08049640 0x000d0 0x000d0 RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

Bref, ce qu'il convient de retenir c'est qu'avant que l'application ne débute la fonction main, le programme ld-linux.so a déjà projeté en mémoire les librairies nécessaires.

2.2.Observer les librairies chargées à l'exécution

La première méthode, et qui est de loin la plus simple, est d'interroger le binaire avec la commande ldd.

Par exemple, la commande

```
fred@kdiv ~ >ldd mtest
      linux-gate.so.1 =>  (0xb781d000)
      libc.so.6 => /lib/libc.so.6 (0xb76b2000)
      /lib/ld-linux.so.2 (0xb781e000)
```

nous informe que le programme mtest dépend de la libc, de linux-gate (dso virtuel, mappé en mémoire pour incorporer des fonctions utilisées par le noyau), et de ld-linux.so, qui est l'interpréteur.

Vous pouvez également utiliser la commande

```
fred@kdiv ~>/lib/ld-linux.so.2 --list  ./mtest
      linux-gate.so.1 =>  (0xb7725000)
      libc.so.6 => /lib/libc.so.6 (0xb75ba000)
      /lib/ld-linux.so.2 (0xb7726000)
```

pour obtenir le même genre de résultats.

La deuxième méthode consiste à inspecter à l'exécution, le mapping mémoire du programme.

Lorsque vous exécutez une commande, un processus (programme s'exécutant) est créé.

Ce processus est identifié par son PID (process identifiant). Il est possible de récupérer depuis un programme son propre pid avec la fonction getpid(). Il est possible depuis le shell, de récupérer le pid de chaque tâche du système.

Par exemple, la commande :

```
fred@kdiv ~ >ps -ejH |tail -n10
 4976  1516  1516 ?        00:00:00     gnome-pty-helpe
 4978  4978  4978 pts/0    00:00:00     bash
 5138  5138  4978 pts/0    00:00:00       man
 5154  5138  4978 pts/0    00:00:00         less
 5312  5312  5312 pts/1    00:00:00     bash
 8930  8930  5312 pts/1    00:00:00       gdb
 8962  8962  5312 pts/1    00:00:00         mtest
 8323  8323  8323 pts/3    00:00:00     bash
10081 10081  8323 pts/3    00:00:00       ps
10082 10081  8323 pts/3    00:00:00       bash
```

va afficher les 10 derniers résultats de listing des processus. La première colonne correspond à l'identifiant de processus.

Si on s'interesse au programme mtest, on constate que son identifiant est 8962. A partir de là, il est très simple de regarder les librairies dynamiques que l'exécution de ce programme a causé.

En effet, le fichier /proc/8192/maps contient le mapping mémoire du programme identifié avec le pid 8192

```
fred@kdiv ~ >cat /proc/8962/maps
08048000-08049000 r-xp 00000000 08:03 3025594    /home/mudflap_test/mtest
08049000-0804a000 rw-p 00000000 08:03 3025594    /home/mudflap_test/mtest
b7e75000-b7e76000 rw-p 00000000 00:00 0
b7e76000-b7fbb000 r-xp 00000000 08:02 221        /lib/libc-2.12.so
b7fbb000-b7fbd000 r--p 00145000 08:02 221        /lib/libc-2.12.so
b7fbd000-b7fbe000 rw-p 00147000 08:02 221        /lib/libc-2.12.so
b7fbe000-b7fc1000 rw-p 00000000 00:00 0
b7fe0000-b7fe1000 rw-p 00000000 00:00 0
b7fe1000-b7fe2000 r-xp 00000000 00:00 0          [vdso]
b7fe2000-b7ffe000 r-xp 00000000 08:02 1054       /lib/ld-2.12.so
b7ffe000-b7fff000 r--p 0001b000 08:02 1054       /lib/ld-2.12.so
b7fff000-b8000000 rw-p 0001c000 08:02 1054       /lib/ld-2.12.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

Les 2 premières lignes correspondent au programme en lui-même, qui a été projeté en mémoire. D'après ce qui est lu, on observe que le programme mtest est projeté deux fois en mémoire puisqu'il y'a deux lignes.

En fait, la première projection va permettre l'utilisation des segments de l'application mtest nécessitant les droits d'exécution, et n'ayant pas de droit d'écriture (r-xp). Typiquement les segments concernés seront ceux contenant les sections de code (.text) et les sections de

variables en read-only.(.rodata)

Il est possible d'obtenir les segments d'un programme avec la commande suivante :

```
fred@kdiv ~ >readelf -l ./mtest
Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version
.gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
```

Le premier segment est le segment .interp. Ce segment contient le chemin d'accés vers le programme ld-linux.so.

Le deuxième segment contient un ensemble de sections nécessitant des droits d'exécution, mais interdisant toute écriture. C'est ce segment qui correspond à la première ligne

Le troisième segment contient un ensemble de sections pouvant être modifié à l'exécution. Par exemple les sections contenant des variables globales.

Les autres seront abordés dans le cours sur le format ELF, et ne sont pas nécessaires pour l'instant.

On peut constater que la libc est projeté en mémoire. Elle est en effet utilisée pratiquement par tous les programmes C. Elle est projetée trois fois pour les mêmes raisons que précédemment.

Idem pour le programme de linkage dynamique ld-linux.so.

Les autres lignes sont moins importantes pour la compréhension, néanmoins :

- la ligne [stack] contient l'adresse de la pile du programme
- la ligne [vdso] est utilisée par le noyau pour utiliser des fonctions spécifiques (exécution d'appel système via kernel_vsyscall et de mise en place de handler de signaux via __kernel_rt_sigreturn)
- il pourrait également yavoir une ligne [heap], laquelle indiquerait le segment mémoire utiliser pour les allocations dynamiques.

En résumé, la deuxième méthode consiste à lire le fichier maps. A partir de là, il est possible d'obtenir plusieurs informations.

2.3.Variables d'environnements.

Le programme ld-linux.so est sensible à plusieurs variables d'environnements qui peuvent modifier son comportement, à l'exécution.

Les différents points ci-dessous vont donner quelques exemples de variables d'environnements utilisées par ld-linux.so, et leur utilité.

2.3.1La variable LD_PRELOAD

Cette variable d'environnement attend une liste de chemin d'accés vers des fichiers de librairies dynamiques.

Par exemple, la commande :

```
export LD_PRELOAD=./libtest1.so:./libtest2.so
```

Lorsqu'un programme va être exécuté, et lorsque les librairies vont être chargées par le programme ld-linux.so, ce dernier va d'abord inspecter le contenu de cette variable d'environnement, avant même de charger les librairies nécessaires au fonctionnement du programme (comme la libc par exemple).

En d'autres termes, les fichiers libtest1.so et libtest2.so seront chargées en mémoire avant la libc.

Par exemple, le programme «example » dépend d'un certain nombre de librairie.

```
fred@kdiv ~/Programmation/divers/sym_resolv >ldd test
      linux-gate.so.1 =>  (0xb77e1000)
      libsym_resolver.so => ./libsym_resolver.so (0xb77dd000)
      libc.so.6 => /lib/libc.so.6 (0xb7673000)
      libdl.so.2 => /lib/libdl.so.2 (0xb766f000)
      /lib/ld-linux.so.2 (0xb77e2000)
```

L'exécution de ce programme va fort logiquement aboutir, via le programme ld-linux.so, à la projection en mémoire de la libc, de la libdl, de la libsym_resolver...

```
fred@kdiv ~ >cat /proc/13850/maps
08048000-08049000 r-xp 00000000 08:03 3538966    /divers/sym_resolv/test
08049000-0804a000 rw-p 00000000 08:03 3538966    /divers/sym_resolv/test
b7e6e000-b7e6f000 rw-p 00000000 00:00 0
b7e6f000-b7e71000 r-xp 00000000 08:02 1055       /lib/libdl-2.12.so
b7e71000-b7e72000 r--p 00001000 08:02 1055       /lib/libdl-2.12.so
b7e72000-b7e73000 rw-p 00002000 08:02 1055       /lib/libdl-2.12.so
b7e73000-b7fb8000 r-xp 00000000 08:02 221        /lib/libc-2.12.so
b7fb8000-b7fba000 r--p 00145000 08:02 221        /lib/libc-2.12.so
b7fba000-b7fbb000 rw-p 00147000 08:02 221        /lib/libc-2.12.so
b7fbb000-b7fbe000 rw-p 00000000 00:00 0
b7fdd000-b7fdf000 r-xp 00000000 08:03 3538962    /divers/libsym_resolver.so
b7fdf000-b7fe0000 rw-p 00001000 08:03 3538962    /divers/libsym_resolver.so
b7fe0000-b7fe1000 rw-p 00000000 00:00 0
b7fe1000-b7fe2000 r-xp 00000000 00:00 0          [vdso]
```

```
b7fe2000-b7ffe000 r-xp 00000000 08:02 1054       /lib/ld-2.12.so
b7ffe000-b7fff000 r--p 0001b000 08:02 1054       /lib/ld-2.12.so
b7fff000-b8000000 rw-p 0001c000 08:02 1054       /lib/ld-2.12.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

La variable LD_PRELOAD va permettre de faire charger en mémoire, et avant tout autre librairie, les fichiers .so de notre choix.

Par exemple, forçons le chargement d'une petite librairie, appelée libdbgprint.so.

La commande qui permet de faire cela est

```
export LD_PRELOAD=./libdbgprint.so
```

Nous allons utiliser la commande strace, qui permet de voir l'ensemble des appels systèmes fait par un programme (et par son chargement). Cette commande va nous permettre de voir dans quelle ordre les librairies sont chargées, lorsque le programme est exécutée.

```
fred@kdiv ~ >strace ./test
execve("./test", ["./test"], [/* 57 vars */]) = 0
brk(0)                                  = 0x89c4000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb772c000
open("./libdbgprint.so", O_RDONLY)      = 3
...
open("/lib/libc.so.6", O_RDONLY)        = 3
...
open("/lib/libdl.so.2", O_RDONLY)       = 3
```

La sortie a volontairement été écourtée (notamment les sections permettant de mapper en mémoire les librairies). Ce qui reste permet néanmoins de constater que la librairie dbgprint est la première à être chargée dynamiquement, avant même la librairie C.

La question est à quoi cela peut-il servir?

La variable LD_PRELOAD est essentiellement utilisée pour surcharger des fonctions définies dans d'autres librairies. Par exemple, en supposant que la librairie libdbgprint.so contient une fonction printf, l'utilisation de la variable LD_PRELOAD permet d'utiliser le printf de la bibliothèque dbgprint plutôt que celle de la libc.

En effet, la librairie dbgprint étant chargée avant, elle est la première analysée lors de la résolution des symboles.

C'est le mécanisme le plus simple pour faire du détournement de fonctions.

2.3.2La variable LD_LIBRARY_PATH

Cette variable permet d'ajouter des répertoires de recherche supplémentaires au programme ld-linux.so. Par défaut, il va chercher les librairies à mapper en mémoire dans les répertoires /lib et /usr/lib. Cettte variable permet d'en spécifier davantage.

Soit, par exemple, le programme mtest. Nous pouvons lister les librairies qui seront chargées en mémoire avec la commande suivante :

```
fred@kdiv ~>ldd mtest
      linux-gate.so.1 =>  (0xb7726000)
      libsym_resolver.so => not found
      libc.so.6 => /lib/libc.so.6 (0xb75bb000)
      libdl.so.2 => /lib/libdl.so.2 (0xb75b7000)
      /lib/ld-linux.so.2 (0xb7727000)
```

Apparemment il y'a un problème avec la librairie libsym_resolver qui ne semble pas pouvoir être localisée...

L'exécution nous confirme cette impression.

```
fred@kdiv ~ >./mtest
./test: error while loading shared libraries: libsym_resolver.so: cannot open
shared object file: No such file or directory
```

Le problème est simple à comprendre : la librairie libsym_resolver.so ne se situant pas dans l'un des répertoires de recherche de ld-linux, elle n'a pu être localisée. La variable LD_LIBRARY_PATH permet de remédier à cela.

```
fred@kdiv ~ >LD_LIBRARY_PATH=. ./test
hello
```

Ce coup-ci le programme s'exécute bien. Nous avons indiqué que le programme ld-linux.so devait également recherché les librairies dans le répertoire courant (LD_LIBRARY_PATH=.)

2.3.3La variable LD_BIND_NOW

Cette variable d'environnement, si elle est non vide, demande à ld-linux.so de résoudre tous les symboles du programme dés l'exécution, plutôt que de le faire au fur et à mesure du besoin.

2.3.4La variable LD_DEBUG/LD_DEBUG_OUTPUT

Cette variable peut être utile pour debugguer l'utilisation de librairie dynamique.

LD_DEBUG attend une valeur lui spécifiant quelles informations de debug devront être affichées à l'exécution. Un moyen de lister les valeurs possibles est le suivant :

```
fred@kdiv ~ >LD_DEBUG=help ./mtest
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

Utilisons la valeur all, pour avoir un exemple de résultat :
fred@kdiv ~/Programmation/Posix/mudflap_test >LD_DEBUG=all ./mtest
     17239:
     17239: file=libc.so.6 [0];  needed by ./mtest [0]
     17239: find library=libc.so.6 [0]; searching
     17239:  search cache=/etc/ld.so.cache
     17239:   trying file=/lib/libc.so.6
 ...
     17239:
     17239: relocation processing: /lib/libc.so.6
...

     17239: symbol=free;  lookup in file=./mtest [0]
     17239: symbol=free;  lookup in file=/lib/libc.so.6 [0]
     17239: binding file /lib/ld-linux.so.2 [0] to /lib/libc.so.6 [0]: normal
symbol `free' [GLIBC_2.0]
     17239:
     17239: calling init: /lib/libc.so.6
     17239:
     17239: symbol=__libc_start_main;  lookup in file=./mtest [0]
     17239: symbol=__libc_start_main;  lookup in file=/lib/libc.so.6 [0]
     17239: binding file ./mtest [0] to /lib/libc.so.6 [0]: normal symbol
`__libc_start_main' [GLIBC_2.0]
     17239:
     17239: initialize program: ./mtest
     17239:
     17239:
     17239: transferring control: ./mtest
     17239:
     17239: symbol=malloc;  lookup in file=./mtest [0]
     17239: symbol=malloc;  lookup in file=/lib/libc.so.6 [0]
     17239: binding file ./mtest [0] to /lib/libc.so.6 [0]: normal symbol
`malloc' [GLIBC_2.0]
     17239: symbol=printf;  lookup in file=./mtest [0]
```

```
     17239: symbol=printf;  lookup in file=/lib/libc.so.6 [0]
     17239: binding file ./mtest [0] to /lib/libc.so.6 [0]: normal symbol
`printf' [GLIBC_2.0]
...
     17239:
     17239: calling fini: ./mtest [0]
     17239:
     17239:
     17239: calling fini: /lib/libc.so.6 [0]
```

Il ya énormément d'informations. Ces informations de debug nous informe des différentes étapes faites par le programme ld-linux.so. Succintement, nous pouvons constater des étapes concernant

- la recherche de la libc, laquelle sera projetée en mémoire
- une phase de relocation, consistant à résoudre les symboles (comme le symbole free) afin de connaître son emplacement en mémoire
- une phase appelant d'éventuels constructeurs
- une phase donnant enfin la main à la fonction main du programme
- une phase de nettoyage à la fermeture du programme, avec appels éventuels de destructeur

Généralement, les sorties sont affichées à l'écran. La variable LD_DEBUG_OUTPUT permet de spécifier un fichier dans lequel sera inscrit les données.

2.3.5La variable LD_PROFILE/LD_PROFILE_OUTPUT

Il est possible de profiler l'exécution d'un programme, afin d'obtenir des statistiques de consommation système des fonctions utilisées par ce dernier.

Cela se fait à l'aide de l'outil gprof.

Il est possible de faire cela également pour les librairies partagées d'un programme. Soit le programme suivant :

```
fred@kdiv ~ >ldd test
      linux-gate.so.1 =>  (0xb7809000)
      libsym_resolver.so => ./libsym_resolver.so (0xb7767000)
      libc.so.6 => /lib/libc.so.6 (0xb769e000)
      libdl.so.2 => /lib/libdl.so.2 (0xb769a000)
      /lib/ld-linux.so.2 (0xb780a000)
```

Ce programme, en plus d'utiliser les librairies habituelles, utilise la libsym_resolver, qui a été récupéré d'internet. Il est possible de faire un profiling d'exécution de la librairie de la manière suivante :

fred@kdiv ~ >export LD_PROFILE=libsym_resolver.so ./test

Cette commande, en plus d'exécuter le programme test, va générer un fichier appelé /var/tmp/libsym_resolver.so.profile.

Ce fichier peut ensuite être décortiqué à l'aide de la commande sprof

```
fred@kdiv ~> sprof  ./libsym_resolver.so /var/tmp/libsym_resolver.so.profile
```

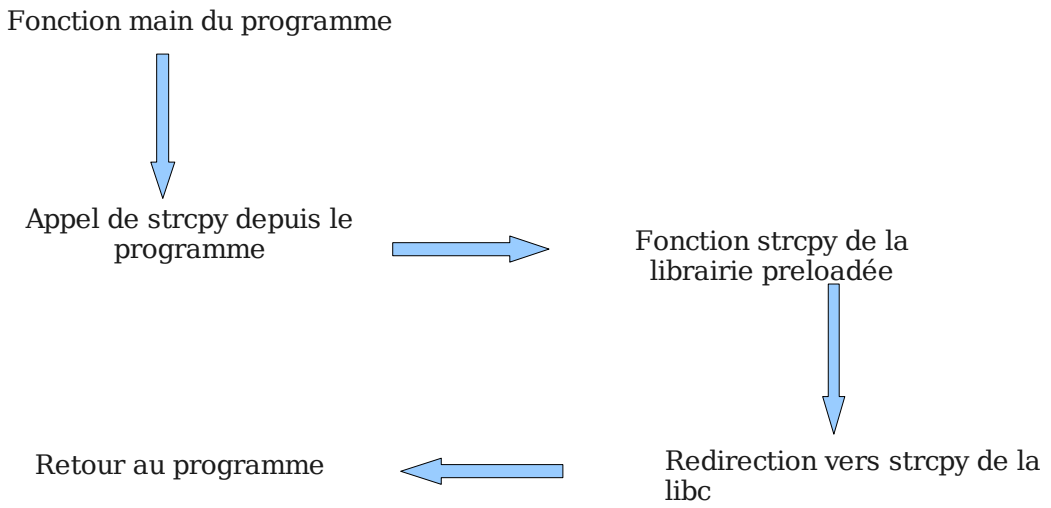
3.Présentation de la libdl

Comme il a été dit en introduction, le programme ld-linux.so, bien que compilé statiquement, a conduit à l'élaboration d'une librairie, appelée libdl.

Cette librairie permet de charger dynamiquement des fichiers partagées, d'y rechercher l'adresse d'un symbole, ou dé recupérer un certain nombre d'information concernant une adresse.

Cette api a plusieurs objectif :

- elle permet de mettre en place le fonctionnement de plugin, en chargeant/déchargeant à l'envie des fichiers .so
- elle permet de surcharger une fonction (à l'aide de LD_PRELOAD). Par exemple, nous pourrions définir une librairie implémentant strcpy et la charger avant la libc à l'aide de LD_PRELOAD. Notre fonction strcpy quant à elle, analyserait les paramètres, avant de les passer à la vraie fonction de la libc.



Pour utiliser la libdl, il faut d'une part inclure le fichier d'entête <dlfcn.h>, et d'autre part compiler le programme (ou la librairie dynamique) avec le support de la libdl, via -ldl.

3.1.Ouvrir un fichier DSO

La première fonction utilisée va permettre d'ouvrir et de projeter en mémoire une librairie dynamique.

```
void *dlopen(const char *filename, int flag);
```

Cette fonction prend en premier paramètre le chemin d'accés vers la librairie dynamiqe à charger, avec la régle suivante :

- si le nom commence par un / ou un ./, le fichier est recherché par analyse du chemin
- si seul le nom est spécifié, alors la fonction recherche le fichier avec les mêmes règles que le programme ld-linux.so

En second paramètre, vous pouvez spécifier d'une part comment les symboles de type fonction doivent être résolus à l'ouverture de la librairie, avec deux choix possibles :

- RTLD_LAZY pour résoudre les symboles au besoin
- RTLD_NOW pour résoudre tous les symboles dés l'ouverture

Il est également possible de préciser la portée des symboles exportés de la librairie.

En valeur de retour est récupérée l'addresse mémoire à laquelle la librairie a été mappée.

En cas d'erreur, la valeur NULL est retournée.

Ci-dessous un exemple mettant en évidence les conséquences d'un appel à la fonction dlopen

```
#include <stdio.h>
#include <dlfcn.h>

int main()
{
    void *p;
    printf("exécuter la commande suivante puis appuyez sur entrée:\n");
    printf("cat /proc/%u/maps\n", getpid());
    getchar();

    p = dlopen("libevent.so", RTLD_LAZY);
    if(p){
        printf("afficher de nouveau le contenu de /proc/%u/maps\n", getpid());
        printf("y'a t'il de nouvelles librairies chargées?\n");
        getchar();
        dlclose(p);
```

```
    } else {
        printf("error : %s\n", dlerror());
    }
    return 0;
}
```

A l'exécution, nous avons deux invitations pour analyser le fichier maps de l'application.

Avant le « dlopen », nous avons le fichier maps suivant :

```
fred ~/ $ cat /proc/18102/maps
08048000-08049000 r-xp 00000000 08:03 2307954
/home/fred/Documents/Cours/cours/linker/test
08049000-0804a000 rw-p 00000000 08:03 2307954
/home/fred/Documents/Cours/cours/linker/test
b764f000-b7650000 rw-p 00000000 00:00 0
b7650000-b7795000 r-xp 00000000 08:01 2000       /lib/libc-2.12.so
b7795000-b7797000 r--p 00145000 08:01 2000       /lib/libc-2.12.so
b7797000-b7798000 rw-p 00147000 08:01 2000       /lib/libc-2.12.so
b7798000-b779b000 rw-p 00000000 00:00 0
b779b000-b779d000 r-xp 00000000 08:01 2017       /lib/libdl-2.12.so
b779d000-b779e000 r--p 00001000 08:01 2017       /lib/libdl-2.12.so
b779e000-b779f000 rw-p 00002000 08:01 2017       /lib/libdl-2.12.so
b77be000-b77c1000 rw-p 00000000 00:00 0
b77c1000-b77c2000 r-xp 00000000 00:00 0          [vdso]
b77c2000-b77de000 r-xp 00000000 08:01 2016       /lib/ld-2.12.so
b77de000-b77df000 r--p 0001b000 08:01 2016       /lib/ld-2.12.so
b77df000-b77e0000 rw-p 0001c000 08:01 2016       /lib/ld-2.12.so
bf888000-bf8a9000 rw-p 00000000 00:00 0          [stack]
```

Après le dlopen, nous constatons que plusieurs librairies, dont la libevent, ont été ajoutées en mémoire. Les autres librairies ont été ajoutées car ce sont des dépendances de la librairie libevent.

```
fred ~/Documents/Cours/cours/linker $ cat /proc/18102/maps
08048000-08049000 r-xp 00000000 08:03 2307954
/home/fred/Documents/Cours/cours/linker/test
08049000-0804a000 rw-p 00000000 08:03 2307954
/home/fred/Documents/Cours/cours/linker/test
081b6000-081d7000 rw-p 00000000 00:00 0          [heap]
b75e8000-b75fd000 r-xp 00000000 08:01 1997       /lib/libpthread-2.12.so
b75fd000-b75fe000 ---p 00015000 08:01 1997       /lib/libpthread-2.12.so
b75fe000-b75ff000 r--p 00015000 08:01 1997       /lib/libpthread-2.12.so
b75ff000-b7600000 rw-p 00016000 08:01 1997       /lib/libpthread-2.12.so
b7600000-b7602000 rw-p 00000000 00:00 0
b7602000-b7613000 r-xp 00000000 08:01 2010       /lib/libresolv-2.12.so
b7613000-b7614000 r--p 00010000 08:01 2010       /lib/libresolv-2.12.so
b7614000-b7615000 rw-p 00011000 08:01 2010       /lib/libresolv-2.12.so
b7615000-b7617000 rw-p 00000000 00:00 0
b7617000-b762a000 r-xp 00000000 08:01 2019       /lib/libnsl-2.12.so
```

```
b762a000-b762b000 r--p 00012000 08:01 2019       /lib/libnsl-2.12.so
b762b000-b762c000 rw-p 00013000 08:01 2019       /lib/libnsl-2.12.so
b762c000-b762e000 rw-p 00000000 00:00 0
b764f000-b7650000 rw-p 00000000 00:00 0
b7650000-b7795000 r-xp 00000000 08:01 2000       /lib/libc-2.12.so
b7795000-b7797000 r--p 00145000 08:01 2000       /lib/libc-2.12.so
b7797000-b7798000 rw-p 00147000 08:01 2000       /lib/libc-2.12.so
b7798000-b779b000 rw-p 00000000 00:00 0
b779b000-b779d000 r-xp 00000000 08:01 2017       /lib/libdl-2.12.so
b779d000-b779e000 r--p 00001000 08:01 2017       /lib/libdl-2.12.so
b779e000-b779f000 rw-p 00002000 08:01 2017       /lib/libdl-2.12.so
b779f000-b77a6000 r-xp 00000000 08:01 1977       /lib/librt-2.12.so
b77a6000-b77a7000 r--p 00006000 08:01 1977       /lib/librt-2.12.so
b77a7000-b77a8000 rw-p 00007000 08:01 1977       /lib/librt-2.12.so
b77a8000-b77bd000 r-xp 00000000 08:01 15699      /usr/lib/libevent-1.4.so.2.2.0
b77bd000-b77be000 rw-p 00015000 08:01 15699      /usr/lib/libevent-1.4.so.2.2.0
b77be000-b77c1000 rw-p 00000000 00:00 0
b77c1000-b77c2000 r-xp 00000000 00:00 0          [vdso]
b77c2000-b77de000 r-xp 00000000 08:01 2016       /lib/ld-2.12.so
b77de000-b77df000 r--p 0001b000 08:01 2016       /lib/ld-2.12.so
b77df000-b77e0000 rw-p 0001c000 08:01 2016       /lib/ld-2.12.so
bf888000-bf8a9000 rw-p 00000000 00:00 0          [stack]
```

3.2.Récupérer un symbole d'un fichier DSO

Une fois projetée en mémoire, il va être intéressant de récupéré des symboles définies par la librairie, afin de les utiliser dans le programme.

La fonction

```
void *dlsym(void *handle, const char *symbol);
```

va lancer la recherche du symbole spécifié en deuxième paramètre, à partir de l'adresse spécifiée en premier paramètre. Cette adresse correspond généralement à la valeur de retour de dlopen. La valeur de retour de dlsym est l'adresse du symbole recherché.

Pour comprendre le fonctionnement de cette fonction, le programme de test suivant va être proposé :

- une librairie dynamique contenant une fonction appelée init est implémentée
- un programme charge cette librairie puis exécute la fonction init de la librairie chargée

```
// fichier pluginhello.c compilée
// de la manière suivante
// gcc -fpic -c pluginhello.c
// gcc pluginhello.o -o libtest.so -shared

#include <stdio.h>


void init(void)
{
                puts("hello plugin");
}


// fichier main.c compilé avec
// gcc main.c -ldl -o main
#include <dlfcn.h>
#include <stdio.h>


int main()
{
                void (*ptrinit)(void) = NULL;
                void *libaddr = dlopen("./libtest.so", 1);
                if(libaddr == NULL){
                                printf("cannot load libtest : %s\n", dlerror());
                                return -1;
                }

                ptrinit = dlsym(libaddr, "init");
                if(ptrinit != NULL){
                                ptrinit();
                }

                dlclose(libaddr);
                return 0;
}
```

L'exécution de ce programme conduit à l'exécution de la fonction init.

3.2.1Etude d'un cas particulier de la fonction dlsym

Comme vous l'avez constaté, le premier paramètre de la fonction dlsym correspond à l'adresse d'un fichier DSO projeté en mémoire. Il existe deux autres valeurs qui peuvent être utilisées :

- RTLD_NEXT

- RTLD_DEFAULT

En utilisant en premier paramètre de dlsym la valeur RTLD_NEXT, la recherche du symbole va débuter à partir de l'adresse mémoire de la bibliothèque en cours.

En utilisant la valeur RTLD_DEFAULT, la recherche du symbole va se faire en recherchant dans l'ordre de chargement des librairies.

La valeur intéressant est évidemment RTLD_NEXT. Elle permet de créer des enveloppes (wrapper) de fonctions, comme le test suivant va le montrer.

L'objectif du programme est de coder une librairie « enveloppant » la fonction strcmp, c'est à dire qu'un appel à cette fonction depuis un programme fait appel à notre librairie, laquelle, après un traitement, va déléguer le travail à la fonction strcmp de la libc.

Voici le programme de test :

```
#include <stdio.h>

void f1(char *const a, char *const b)
{
        printf("%d\n", strcmp(a, b));
}

int main()
{
        char a[] = "hello";
        char b[] = "world";

        f1(a, b);
        return 0;
}
```

L'exécution de ce programme va se faire sans surprise, de même que le résultat.

```
fred ~/ $ ./test
-1
```

Voici maintenant le code de la librairie :

```
#include <dlfcn.h>
#include <stdio.h>

int strcmp(const char *c1, const char *c2)
{
        static int (*true_strcmp)(const char*, const char*) = NULL;
        if(true_strcmp == NULL){
                        true_strcmp = dlsym(RTLD_NEXT, "strcmp");
        }
```

```
        printf("param1=%s; param2=%s\n", c1, c2);
        if(true_strcmp){
                        return true_strcmp(c1, c2);
        }

        return 0;
}
```

Cette librairie se compile de la manière suivante :

```
fred ~/ $ gcc -fpic  correction.c -o libtest.so -shared -ldl
```

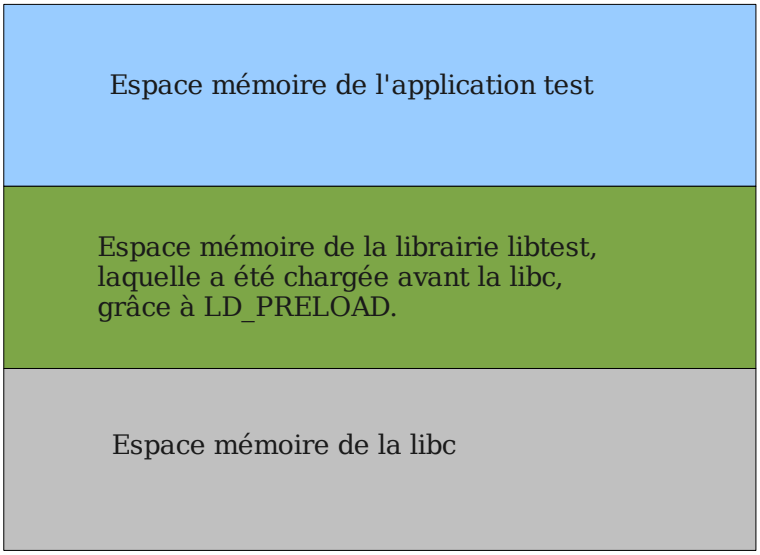
Nous allons reéxécuter le programme ./test, en demandant au programme ld-linux.so de charger en priorité la librairie libtest.so

```
fred ~/ $ LD_PRELOAD=./libtest.so ./test
param1=hello; param2=world
-1
```

Nous constatons une différence dans l'exécution. En effet, la fonction strcmp utilisée par le programme est désormais dans la librairie libtest.so. Cette dernière affiche les valeurs des variables, puis délégue le travail au vrai strcmp.

En l'occurence, le « vrai » strcmp se situe dans la libc, qui a été chargée après la libtest.

Schématiquement, le mapping mémoire est le suivant



Un appel à la fonction strcmp depuis l'application « test » va utiliser le symbole de la librairie libtest.

A partir de la librairie libtest, nous lançons la recherche du prochain (RTLD_NEXT) symbole de nom « strcmp » que l'on peut trouver en mémoire.

En l'occurrence, ce dernier se situera dans la libc.

A retenir donc : il existe deux valeurs prédéfinies que l'on peut utiliser avec la fonction dlsym. L'utilisation de la valeur RTLD_NEXT est utilisée par emballer une fonction. Et enfin, l'utilisation de RTLD_NEXT ou RTLD_DEFAULT rend inutile l'utilisation de dlopen, puisque la recherche va se faire à partir de l'adresse de la librairie courante.

3.3.Libérer les ressources.

Dans le cas où nous avons projeté une librairie en mémoire via dlopen, il est recommandé de libérer de l'espace mémoire lorsque les symboles qu'elle definit ne sont plus utilisés.

La fonction à utiliser est dlclose:

```
int dlclose(void *handle);
```

3.4.Récupération d'informations sur l'adresse d'une fonction

Il existe certaines situations dans lesquelles nous avons à notre disposition l'adresse (approximative) d'une fonction, sans connaître son nom, ni son emplacement. C'est le cas notamment lorsqu'on utilise la fonction backtrace, qui retourne la pile d'appel sous forme de tableau contenant les adresses des fonctions.

La fonction dladdr, présentée avec un exemple ci-dessous, permet de récupérer des informations sur une adresse de fonction, à la condition que celle-ci soit définie dans une librairie dynamique (n'oublions pas que la libdl traite principalement tout ce qui concerne la liaison dynamique).

Voici un exemple récupérant des informations sur la fonction malloc (qui appartient à la librairie partagée libc.so)

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
        // recupération de l'adresse de malloc (pour l'exemple)
        void *exemple = malloc;
        Dl_info info;

        if(dladdr(exemple, &info) == 0){
                printf("error %s\n", dlerror());
                return 0;
        }

        printf("symbole définit dans le fichier %s\n", info.dli_fname);
        printf("nom du symbole  %s\n", info.dli_sname);
        return 0;
}
```

La sortie du programme est la suivante :

```
fred ~/ $ ./test
symbole définit dans le fichier ./test
nom du symbole  malloc
```

4.Idées d'exercices

L'objectif de cet exercice est de créer une bibliothèque libdemul. L'exercice fera appel à vos connaissances concernant :

- les extensions de gcc
- le linkage dynamique
- la compilation de librairies

L'exercice sera fera par étapes. Le but est de créer des surcouches à certaines fonctions afin d'en modifier le comportement.

- Créer un constructeur de librairie initialisant une variable globale statique de type uint64_t avec la valeur zéro.
- Créer une fonction surcouchant la fonction write (dont vous avez le prototype par l'exercice précédent). La fonction write prend en premier paramètre un descripteur de fichier, identifiant là où doivent être écrites les données. Par exemple, dans le cas d'un affichage à l'écran, le descripteur de fichier vaut la valeur 1 (valeur constante). Dans le cas d'un fichier standard, le numéro de descripteur peut varier entre 3 et1048576 (limite système). Votre surcouche devra permettre, dans le cas d'une écriture dans un fichier standard, d'afficher à l'écran ce qui va être écrit. Par exemple, l'appel suivant :

ssize_t ret = write(24, "coucou", 6);
devra écrire coucou dans le descripteur de fichier 24 (comportement normal) et en plus afficher coucou à l'écran. C'est exactement ce que permet la commande "tee" sous linux. Attention, dans le cas d'une écriture sur le terminal (correspondant à un appel write(1, "coucou", 6)) le message ne devra être affiché qu'une fois.

Vous incrémenterez la variable globale statique avec le nombre de caractère qui ont été écrit sur le terminal

- Créer une destructeur de librairie qui permettra d'afficher le nombre d'octets affichés depuis le début du programme.
- Donnez la(les) commandes permettant de compiler cette librairie
- Donnez un exemple d'utilisation de cette librairie sur un programme appelé "test1"