Jouer avec la « libdl »

Yann Langlais

En 1992, alors que je découvrais les joies de l'édition de texte avec Emacs, une question m'est venue à l'esprit. Pourquoi diable les extensions d'Emacs ont-elles été écrites en Elisp (Emacs-lisp) et non en C ? La réponse était simple. Il n'existait pas alors de possibilité standard en C de charger et décharger des bibliothèques en cours d'exécution.

Quelques années plus tard, et bien qu'Emacs ait été remplacé par Vi dans mes habitudes dactylographiques, je découvris avec plaisir un outil qui rendait caduque mon explication. Je venais de tomber par hasard sur dlfcn.h.

Ce fichier d'en-tête définit les prototypes de points d'entrée forts appréciables et permettant de charger et décharger des bibliothèques et de récupérer des points d'entrée ou des symboles (variables) pour les utiliser à volonté.

Aujourd'hui, et bien que ces fonctions soient à la base de la notion des « plug-in » en C et C++, leur utilisation me semble encore trop restreinte. Voyons à travers quelques exemples ce que nous pouvons en faire.

Les exemples fournis n'ont pas pour vocation d'être toujours justes ni d'effectuer tous les tests appropriés. Ils n'ont valeur d'exemple que pour le domaine très restreint de l'illustration du propos. En temps normal, il est en particulier nécessaire de tester toutes les valeurs retour de `dlopen` et `dlsym` avant de les utiliser.

L'article est basé sur Solaris et Linux.

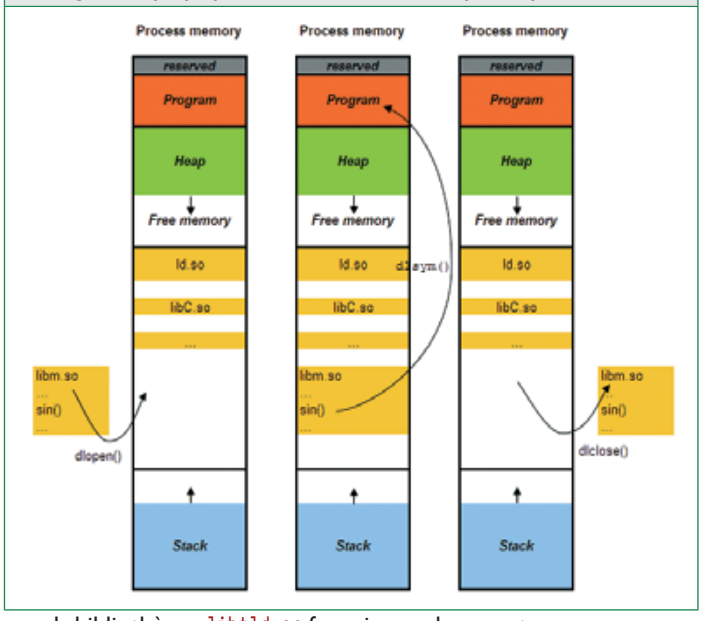
1. Présentation de la libdl.so

1.1 Présentation générale

La `libdl.so` permet de charger (et de décharger) des bibliothèques partagées pendant l'exécution d'un programme et de mettre à disposition du programme tout ou partie des symboles (noms de fonctions ou de variables globales) chargés. C'est un éditeur de liens en cours d'exécution. Il effectue les mêmes tâches que `ld` lors de l'étape de l'édition de liens (figure 1).

Cette bibliothèque est intimement liée au format de fichiers binaires ELF **[1]** (*Extecutable and Linkable Format*) qui a permis d'obtenir du code librement déplaçable dans l'espace mémoire des processus. Il existe des plateformes qui ne supportent pas le format ELF. Il existe aussi des plateformes qui, bien que supportant les bibliothèques partagées, n'implémentent pas la `libdl`. Pour une partie de ces plateformes, il existe cependant une alternative

Figure 1 : Synoptique du fonctionnement de dlopen, dlsym et dlclose



avec la bibliothèque `libtld.so` fournie avec le paquet `libtool` **[2]** de GNU. Les points d'entrée de la `libdlt.so` sont similaires à ceux de la `libdl.so`, mais sont préfixés par « `lt_` ».

Les variations entre les différentes plateformes viennent du relatif jeune âge de cette bibliothèque et donc de sa standardisation récente et imparfaite.

Initialement créée par SUN, la norme POSIX 1003.1 **[3]** a standardisé les fonctions `dlclose`, `dlerror`, `dlopen` et `dlsym`. Ces fonctions sont aussi présentes dans la Linux Standard Base (LSB) version 1.3 **[4]**. Sun a, par la suite, ajouté la fonction `dladdr`. GNU l'a reprise et a ajouté `dlvsym`.

Les différents drapeaux `RTLD_*` ne sont pas non plus tous standardisés, mais, au moins sur Solaris et Linux, les valeurs engendrent des comportements assez proches les uns des autres (à une exception importante près que nous verrons au paragraphe « Chaînage de fonctions »).

1.2 Description des points d'entrée

La bibliothèque contient 5 points d'entrée principaux : `dlopen`, `dlsym`, `dlclose`, `dlerror` et `dladdr`.

1.2.1 dlopen

```
extern void *   dlopen(const char *, int);
```

Cette fonction charge en mémoire la bibliothèque désignée par la chaîne de caractère passée en premier paramètre. Le second paramètre est un drapeau qui fixe le mode de fonctionnement de l'éditeur de liens à la volée. Ce second peut prendre les valeurs (documentées) suivantes :

- RTLD_LAZY, pour ne résoudre les symboles de la bibliothèque à ouvrir que lorsque ceux-ci sont explicitement demandés ;
- RTLD_NOW, pour résoudre tous les symboles lors du chargement de la bibliothèque.

En complément de ces valeurs, il existe une co-valeur RTLD_GLOBAL augmentant la portée des symboles en les rendant disponibles externes (i. e. non préfixés en C par static) pour les bibliothèques chargées par la suite. Elle s'emploie en l'accolant par un ou binaire « | » à l'une des valeurs possibles du drapeau : RTLD_LAZY | RTLD_GLOBAL ou RTLD_NOW | RTLD_GLOBAL.

dlopen retourne un pointeur vers une poignée (*handle*) caractéristique de la bibliothèque sous forme d'un pointeur générique (void *). Celui-ci sera nul en cas d'échec.

Notons que si dlopen renvoie NULL, la variable globale errno ne devrait pas avoir été modifiée et que strerr() ne pourra donc fournir d'explication correcte. Il faudra pour cela appeler dlerror().

1.2.2 dlsym

```
extern void *   dlsym(void *, const char *);
```

dlsym se charge de retrouver (résoudre) l'adresse d'un symbole dont le nom est passé en paramètre.

Le premier paramètre passé est un pointeur vers la « poignée » retournée par dlopen.

Il existe en standard deux poignées particulières définies lorsque _GNU_SOURCE est définie. Ces poignées sont définies par les symboles RTLD_DEFAULT et RTLD_NEXT.

RTLD_DEFAULT précise à dlsym qu'il faut rechercher la première occurrence du symbole dans les bibliothèques déjà chargées en mémoire, dans l'ordre de leur chargement.

RTLD_NEXT stipule à dlsym qu'il lui faut chercher l'occurrence suivante du symbole dans les bibliothèques suivant la bibliothèque en cours. Ce pseudo-pointeur ne fonctionne que pour les bibliothèques partagées.

Suivant les implémentations, on pourra aussi retrouver le symbole RTLD_SELF qui recherchera le symbole dans la bibliothèque en cours.

Le second paramètre est le symbole à rechercher.

dlsym retourne le pointeur sur le symbole demandé, ou, si celui-ci n'est pas retrouvé, un pointeur NULL.

ATTENTION

dlsym() ne permet pas de résoudre que des noms de fonction, mais tous les symboles en général. On peut donc aussi retrouver des variables globales (i. e. variables définies hors fonctions et sans l'attribut static).

1.2.3 dlclose

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

La fonction dlclose() a pour fonction d'éliminer les liens et de décharger des bibliothèques chargées par dlopen().

Elle prend en argument la poignée fournie par dlopen().

Elle retourne 0 quand tout ce passe bien ou un code d'erreur positif en cas de soucis.

1.2.4 dlerror

```
extern char *   dlerror(void);
```

dlerror() retourne une chaîne de caractères contenant le dernier message d'erreur généré par la libdl ou NULL si aucune erreur n'est survenue.

Après l'appel, dlerror() remet les codes d'erreur de la bibliothèque à 0. En conséquence, un second appel à dlerror suivant immédiatement le premier renverra NULL.

Les chaînes de caractères retournées ne doivent pas être libérées par free().

1.2.5 dladdr

```
extern int      dladdr(void *, Dl_info *);
```

dladdr() détermine si une adresse est localisée dans un objet en mémoire. Si tel est le cas, il retourne des informations relatives à cet objet.

dladdr() prend en premier paramètre une adresse sous la forme d'un pointeur anonyme (void *). C'est cette adresse qu'il essaiera de retrouver parmi les segments de texte et de données présents en mémoire. Le second paramètre est un pointeur vers une structure allouée de type DL_info qui sert de réceptacle aux informations de retour. La DL_info contient les champs suivants :

```
typedef struct       dl_info {
          char                *dli_fname;
          void                *dli_fbase;
          char                *dli_sname;
          void                *dli_saddr;
} Dl_info;
```

dli_fname contient le nom du fichier dans lequel est stocké le pointeur fourni en argument ; dli_fbase contient l'adresse de base du fichier en mémoire ; dli_sname contient le nom du fichier source du bloc correspondant au pointeur fourni si les informations de *debug* sont présentes. dli_sbase contient l'adresse du symbole contenant le pointeur fourni.

dladdr() retourne 0 s'il ne trouve pas d'objet contenant le pointeur passé. Toute autre valeur que 0 signifie que des informations ont été trouvées et que la structure DL_info a été remplie.

Pour de plus amples informations sur ces points d'entrée, il est nécessaire de s'en remettre aux manuels **[5]**.

D'autres informations concernant la programmation de bibliothèques partagées peuvent être trouvées en référence **[6]**.

Enfin, certaines variables d'environnement influent sur les bibliothèques, ou plutôt sur l'éditeur de liens **[7]**. L'une d'entre elles sera particulièrement utile dans les paragraphes suivants.

1.2.6 Constructeur et destructeur

Il est parfois nécessaire d'accomplir certaines actions lors du chargement ou du déchargement d'une bibliothèque. On peut, par exemple, devoir réserver de la mémoire et initialiser certaines variables lors du chargement, et libérer la mémoire avant le déchargement.

Pour ce faire, il n'y a malheureusement pas de norme, mais deux écoles distinctes. Il y a, tout d'abord, la méthode « historique » qui est de définir deux fonctions `void _init()` et `void _fini()`. Ces 2 points d'entrée sont, en général, créés automatiquement par les différents *linkers*. Le fait de les redéfinir dans nos bibliothèques et de demander au linker d'éviter de les générer permet donc de remplacer les fonctions standards par les nôtres. Seulement, en général, les éditeurs de liens ne génèrent pas des fonctions vides et le code manquant risque d'avoir des effets secondaires indésirables. Du reste, si Linux supporte encore la fonctionnalité en tant qu'« obsolète », Sun l'interdit (il ne permet pas de demander au compilateur de ne pas générer la fonction standard). Il est donc fortement recommandé d'éviter l'utilisation de ces fonctions.

La seconde méthode revient à utiliser des directives de compilation propriétaires. Par exemple, si l'on veut que la fonction `void onload(void)` soit appelée au chargement de la bibliothèque, et `void onunload(void)`, on écrira sous Linux (compilateur gcc) dans le code source :

```
void __attribute__ ((constructor)) onload(void) {
          printf("loading\n");
}

void __attribute__ ((destructor)) onunload(void) {
          printf("unloading\n");
}
```

et sous Sun (compilateur Forte 6) :

```
#pragma init (onload)
void onload(void) {
          printf("loading\n");
}

#pragma fini (onunload)
void onload(void) {
          printf("unloading\n");
}
```

Il est clair que si l'on recherche la portabilité, il est nécessaire de faire en sorte de se passer de ce genre de chose, à moins d'utiliser les `#ifdef`.

2. Un exemple simple

Voyons l'utilisation de la `libdl` à travers un petit exemple simple.

Admettons que nous voulions écrire un programme qui fournit le sinus d'une liste de valeurs passées en paramètre. Admettons de plus que nous n'ajoutions pas la bibliothèque `libm` à la compilation.

Nous allons donc charger notre bibliothèque « manuellement » pendant l'exécution du programme.

```
cat > sin.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
#include <stdlib.h>
int main(int n, char *a[]) {
    int i;
    void *lib;
    double (*sin)(double);

    /* load /usr/lib/libm.so into memory: */
    if (!(lib = dlopen("/usr/lib/libm.so", RTLD_
LAZY)))
        return 1;

    /* resolve symbole "sin": */
    if (!(sin = (double (*)(double)) dlsym(lib,
"sin")))
        return 2;

    /* print sine of all arguments: */
    for (i = 1; i < n; i++) printf("sin(%f) = %f\n",
        atof(a[i]), sin(atof(a[i])));

    /* unload /usr/lib/libm.so */
    dlclose(lib);

    return 0;
}
EOF
% gcc sin.c -o sin -ldl
```

Le lancement du programme `sin` donne :

```
% sin 0 0.22 .5 1 3.14159
sin(0.000000) = 0.000000
sin(0.220000) = 0.218230
sin(0.500000) = 0.479426
sin(1.000000) = 0.841471
sin(3.141590) = 0.000003
```

Nous pouvons aussi rendre plus « générique » notre programme en lui faisant calculer autre chose que des sinus. Nous pouvons, par exemple, avec un unique exécutable, faire calculer de nombreuses fonctions mathématiques.

Pour ce faire, nous pouvons, par exemple, tenter de charger la fonction qui porte le même nom que le programme (argument 0). Ainsi, en renommant le programme ou en faisant des liens symboliques vers d'autres symboles, notre programme aura des comportements différents :

```
% cat > math.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
#include <stdlib.h>
int main(int n, char *a[]) {
    int i;
    void *lib;
    double (*math)(double);

    /* load /usr/lib/libm.so into memory: */
    if (!(lib  = dlopen("/usr/lib/libm.so", RTLD_LAZY)))
        return 1;
    /* resolve symbole a[0]: */
    if (!(math = (double (*)(double)) dlsym(lib, a[0])))
        return 2;
    /* call a[0] for all command line agruments: */
    for (i = 1; i < n; i++) printf("%s(%f) = %f\n",
        a[0], atof(a[i]), math(atof(a[i])));
    /* unload libm: */
    dlclose(lib);

    return 0;
}
EOF
% gcc math.c -o sin -ldl
```

et voyons le résultat :

```
% sin 0 0.22 .5 1 3.14159
sin(0.000000) = 0.000000
sin(0.220000) = 0.218230
sin(0.500000) = 0.479426
sin(1.000000) = 0.841471
sin(3.141590) = 0.000003
% mv sin cos
% cos 0 0.22 .5 1 3.14159
cos(0.000000) = 1.000000
cos(0.220000) = 0.975897
cos(0.500000) = 0.877583
cos(1.000000) = 0.540302
cos(3.141590) = -1.000000
% ln -s cos tan
tan 0.22 .5 1 3.14159
tan(0.220000) = 0.223619
tan(0.500000) = 0.546302
tan(1.000000) = 1.557408
tan(3.141590) = -0.000003
% mv tan log
log 1 2.718282 10
log(1.000000) = 0.000000
log(2.718282) = 1.000000
log(10.000000) = 2.302585
% cp cos sqrt
sqrt 2 4 9
sqrt(2.000000) = 1.414214
sqrt(4.000000) = 2.000000
sqrt(9.000000) = 3.000000
```

Bien sûr, cet exemple ne fonctionne que pour les fonctions de la bibliothèque mathématique `libm` qui ont pour prototype `double (*)(double)`. Si l'on renommait notre programme original en `pow`, le résultat serait tout à fait aléatoire : la fonction `pow` de la `libm` ira chercher un second argument sur la pile et prendra ce qu'il trouvera.

3. Portée des symboles : le privé et le public

La portée des symboles d'une bibliothèque varie en fonction de la manière dont sont définis les symboles. Seuls les symboles définis en dehors d'une fonction sont accessibles de l'extérieur. La notion de « portée » est déterminée par les mots `static` et `extern` (ou rien). Voyons cela à travers un exemple :

```
% cat > libtest.c <<EOF
static int static_int = 55;
       int extern_int = 66;

static void static_function() {
    extern_int--;
}
void extern_function() {
    extern_int++;
}
EOF
% gcc -G -shared -fPIC libtest.c -o libtest.so
% cat > portee.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
int main(void) {

    void *h;
    void *p;
    if (!(h = dlopen("libtest.so", RTLD_LAZY))) {
        fprintf(stderr, "error loading libtest.so: %s\n",
                dlerror());
        return 1;
    }
    if (!(p = dlsym(h, "static_int")))
        fprintf(stderr, "error resolving static_int: %s\n",
                dlerror());
    else
        printf("libtest.static_int is at %x "
                "and its value is %d\n", p, *(int *) p);

    if (!(p = dlsym(h, "extern_int")))
        fprintf(stderr, "error resolving extern_int: %s\n",
                dlerror());
    else
        printf("libtest.extern_int is at %x "
                "and its value is %d\n", p, *(int *) p);

    if (!(p = dlsym(h, "static_function")))
        fprintf(stderr,
                "error resolving static_function: %s\n",
                dlerror());
    else
        printf("libtest.static_function is at %x\n", p);

    if (!(p = dlsym(h, "extern_function")))
        fprintf(stderr,
                "error resolving extern_function: %s\n",
                dlerror());
    else
        printf("libtest.error_function is at %x\n", p);

        dlclose(h);
    return 0;
}
EOF
% cc -o portee portee.c -ldl
```

Vérifions tout d'abord la portée (`Bind`) des symboles avec `nm` (sous Solaris, le résultat de `nm` sous Linux est un peu moins parlant) :

```
% nm libtest.so | egrep \
"Index|static_int|extern_int|static_function|extern_function"
[Index]  Value Size Type  Bind  Other Shndx  Name
[41]     |  672|  36|FUNC |GLOB |0    |5     |extern_function
[42]     |66404|   4|OBJT |GLOB |0    |10    |extern_int
[31]     |  616|  36|FUNC |LOCL |0    |5     |static_function
[30]     |66408|   4|OBJT |LOCL |0    |10    |static_int
```

On voit clairement que les symboles préfixés par `static_` et définis localement n'ont qu'une portée locale (`Bind` vaut `LOCL` par opposition à `GLOB` pour « globale »). Voilà ce que l'on obtient en lançant le programme de test :

```
% portee
error resolving static_int: ld.so.1:
 portee: fatal: static_int: can't find symbol
libtest.extern_int is at ff270364 and its value is 66
error resolving static_function: ld.so.1:
 portee: fatal: static_function: can't find symbol
libtest.error_function is at ff2602a0
```

Dans cet exemple, la seule manière d'accéder à `static_int` depuis notre programme de test serait d'ajouter des fonctions spécifiques :

```
% cat > libtest2.c <<EOF
static int static_int = 55;
       int extern_int = 66;
```

```
static void static_function() {
    extern_int--;
}

void extern_function() {
    extern_int++;
}
int  static_int_get() {
    return static_int;
}
int  static_int_set(int i) {
    return static_int = i;
}
void static_int_print() {
    printf("static_int = %d\n", static_int);
}
EOF
% gcc -G -shared -fPIC libtest2.c -o libtest2.so
% cat > portee2.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
int main(void) {
    void *h;
    void *p;
    int  (*get)();
    int  (*set)();
    void  (*print)();

        if (!(h = dlopen("libtest2.so", RTLD_LAZY))) {
        fprintf(stderr, "error loading libtest2.so: %s\n",
                dlerror());
        return 1;
    }
    if (!(get = (int (*)()) dlsym(h, "static_int_get")))
        fprintf(stderr, "error resolving static_int: %s\n",
                dlerror());
    else
        printf("static_int_get() returns %d\n", get());

    if (!(set = (int (*)(int)) dlsym(h, "static_int_set")))
        fprintf(stderr,
                "error resolving static_int_set: %s\n",
                dlerror());
    else
        printf("setting static_int with "
                "static_int_get() to 33\n", set(33));

    if (!(print = (void (*)()) dlsym(h, "static_int_print")))
        fprintf(stderr,
               "error resolving static_int_print: %s\n",
                dlerror());
    else {
        printf("call static_int_print() : ");
        print();
    }
    dlclose(h);
    return 0;
}
EOF
% gcc portee2.c -o portee2 -ldl
```

Le nouveau programme de test donne :

```
% portee2
static_int_get() returns 55
setting static_int with static_int_get() to 33
call static_int_print() : static_int = 33
```

Ce genre de restrictions et ces formes d'appels ressemblent bien à celle que l'on peut avoir avec les attributs `private` ou `public` du C++.

4. Vers l'auto-programmation

Le mythe de l'auto-programmation n'est pas récent. Dans la philosophie du `lisp`, par exemple, le programme est une liste et la liste est [potentiellement] un programme. Tout programme est donc capable de se créer des extensions. En règle générale, les langages de script sont des instruments de choix pour l'auto-programmation ou la génération de code spécifique conditionnel et son évaluation à la volée :

```
#!/bin/zsh
cmd="ls -alrt $1"
eval $cmd
```

Le BASIC des années 80 avait sa commande `merge` qui permettait de créer du code dans un fichier et de le réinjecter par la suite dans le programme en cours. De nombreux programmeurs ont un peu été déroutés par l'absence d'équivalent en C. Mais, à l'étape de compilation près, les fonctions `dl*` sont maintenant disponibles pour mimer ce fonctionnement :

```
% cat > dltest.c <<EOF
#include <stdlib.h>
#include <stdio.h>
#include <sys/fcntl.h>
#include <string.h>
#include <dlfcn.h>
int main(void) {
    int fd;
    void *lib;
    void (*funct)();
    char code[] = "#include <stdio.h>\n\
void hello() { printf(\"hello, world\\\n\"); }\n";
    /* Create hello.c */
    if ((fd = open("./hello.c",
                O_WRONLY | O_CREAT, 0640)) < 1) {
        perror("cannot open \"hello.c\"");
        return 1;
    }
    write(fd, code, strlen(code));
    close(fd);
    /* compile hello.c */
    if (
    system("gcc -shared -fPIC -G hello.c -o hello.so") < 0)
        return 2;
    /* read hello.so */
    if (!((lib = dlopen("./hello.so", RTLD_LAZY))))
        return 3;
    /* retrieve "hello" symbol pointer */
    if (!(funct = (void (*)()) dlsym(lib, "hello"))) {
        dlclose(lib);
        return 4;
    }
    /* call funct() */
    funct();
    dlclose(lib);
    return 0;
}
EOF
% gcc dltest.c -o dltest -ldl
```

ATTENTION

Si vous recopiez ce code à la main ou avec un copier/coller, il faut éliminer un des 3 « \ » dans la ligne :
`void hello() { printf(\"hello, world\\\n\"); }\n";`

5. Détournements et surcharge

La `libdl` permet aussi de faire des détournements de fonctions.

Admettons que nous voulions afficher un message à chaque `malloc()` et à chaque `free()`. Il nous suffit de réécrire nos deux fonctions et de les faire charger en lieu et place des points d'entrée système par un petit tour de passe-passe. Cependant, il nous faut aussi appeler les fonctions du système afin d'effectuer les opérations d'allocation ou de désallocation.

Le premier tour de passe-passe (le chargement de notre code au lieu du code système) peut se faire de deux façons :

- en recompilant le programme cible et en forçant l'éditeur de liens à utiliser notre bibliothèque ;
- en demandant le préchargement de notre bibliothèque (nettement plus élégant, et *presque* toujours possible).

Concentrons-nous sur la seconde possibilité. Il nous suffit, avant lancement du programme cible, de renseigner la variable `LD_PRELOAD` :

```
# en sh/ksh/bash/zsh ...:
LD_PRELOAD=malib.so programme_cible
# en csh/tcsh:
(setenv LD_PRELOAD malib.so; programme_cible)
```

Prenons un programme qui ne fait que charger une bibliothèque dynamique et attendre :

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

int main(void) {
    void *p;
    p = dlopen("lib1.so", RTLD_LAZY);
    pause();
    return 0;
}
```

Un programme dépend explicitement de bibliothèques, comme la libc, par exemple. La liste de ces dépendances peut être visualisée grâce à la commande `ldd`.

```
% ldd pause
```

donnera sur Solaris :

```
        libdl.so.1 =>    /usr/lib/libdl.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        /usr/platform/SUNW,Sun-Fire/lib/libc_psr.so.1
```

En temps normal, dès que le code exécutable d'un programme est chargé en mémoire, les dépendances sont chargées les unes après les autres. Chaque bibliothèque pouvant elle-même dépendre d'autres bibliothèques. Dans le cas d'une bibliothèque chargée par `dlopen()`, ici `lib1.so`, nous avons pendant l'exécution (commande `pldd` sous Solaris) :

```
/usr/lib/libdl.so.1
/usr/lib/libc.so.1
/usr/platform/sun4u/lib/libc_psr.so.1
lib1.so
```

La variable d'environnement indique aux routines de chargement des programmes de charger le contenu de la variable en mémoire juste après le chargement du segment exécutable et juste avant le chargement de la première bibliothèque dans la liste des dépendances.

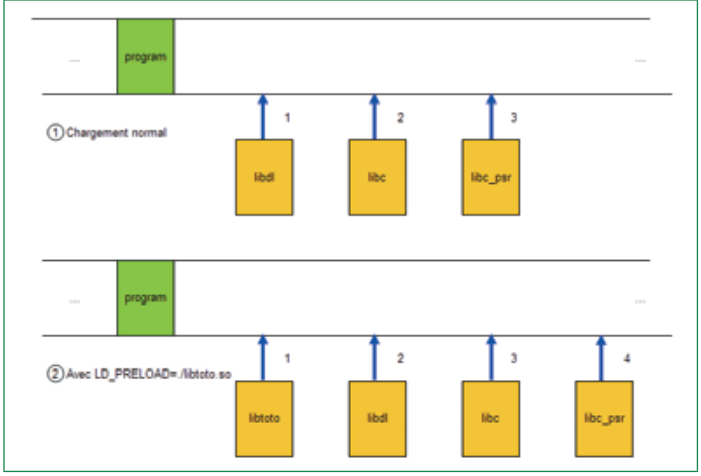
En lançant la commande :

```
% LD_PRELOAD=lib2.so pause
```

On obtient :

```
lib2.so
/usr/lib/libdl.so.1
/usr/lib/libc.so.1
/usr/platform/sun4u/lib/libc_psr.so.1
lib1.so
```

Figure 2 : Chargement normal d'un programme (1) et chargement avec LD_PRELOAD (2)



Supposons maintenant que notre programme cible `tst_alloc` soit le suivant :

```
#include <stdlib.h>

int main() {
    char *str;
    if (!(str = (char *) malloc(32))) return 1;
    free(str);
    return 0;
}
```

que nous compilons de la manière suivante :

```
% gcc -o tst_alloc tst_alloc.c
```

Supposons maintenant le code `mymalloc.c` suivant que nous voulons exécuter :

```
#include <stdio.h>
void * malloc(size_t size) {
    printf("malloc(%d)\n", size);
    return NULL;
}
void free(void *p) {
    printf("free(%x)\n", p);
}
```

que nous compilerons de la façon suivante afin d'en faire une bibliothèque :

```
% gcc -shared -fPIC -o libmymalloc.so mymalloc.c
```

Le lancement de `tst_malloc` seul s'effectue (normalement) sans erreur ni message et retournera 0 (la variable `$?` vaudra 0). Insérons maintenant notre bibliothèque :

```
% LD_PRELOAD=libmymalloc.so tst_malloc
```

affichera :

```
malloc(32)
```

de plus, la commande `echo $?` retournera 1, ce qui est tout à fait normal, puisque notre fonction `malloc` retournera le pointeur 0 et n'effectuera pas l'allocation.

Nous avons surchargé dynamiquement la fonction standard d'allocation mémoire en insérant la nôtre.

Voyons maintenant comment appeler le VRAI `malloc` au sein de notre `malloc`. Pour cela, penchons-nous sur l'aide de `dlsym()` (`man dlsym`). Nous voyons que certains *defines* peuvent remplacer le pointeur vers la bibliothèque (handle). En particulier, `RTLD_NEXT`. L'appel à `dlsym(RTLD_NEXT, symbol)` va rechercher le point d'entrée `symbol` dans la bibliothèque suivante (par rapport à celle en cours).

Notre bibliothèque `libmymalloc.so` va donc se compliquer de la sorte :

```
#include <stdio.h>
#include <dlfcn.h>
void * malloc(size_t size) {
    static void *(*sys_malloc)(size_t) = NULL;
    if (!sys_malloc) {
        if (!(sys_malloc =
            (void *(*)(size_t))dlsym(RTLD_NEXT,"malloc"))) {
            perror("cannot fetch system malloc\n");
            exit(1);
        }
    }
    printf("malloc(%d)\n", size);
    return sys_malloc(size);
}
void free(void *p) {
    static void (*sys_free)(void *) = NULL;
    if (!sys_free) {
        if (!(sys_free =
            (void (*)(void *)) dlsym(RTLD_NEXT, "free")))
{
            perror("cannot fetch system free\n");
            exit(2);
        }
    }
    printf("free(%x)\n", p);
    sys_free(p);
}
```

L'étape de compilation de la bibliothèque se voit aussi légèrement modifiée :

```
% gcc -shared -fPIC -G -o \
 libmymalloc.so mymalloc.c -ldl -D_GNU_SOURCE
```

La commande

```
% LD_PRELOAD=libmymalloc.so tst_malloc
```

affichera :

```
malloc(32)
free(20888)
```

(l'adresse en paramètre de `free` est variable en fonction de l'architecture, entre autres) et la variable `$?` devrait maintenant être à 0.

De nombreuses applications de cette technique sont possibles. Elles peuvent aller du débogueur mémoire à l'écriture de chevaux de Troie ou portes dérobées... Certaines restrictions sont cependant tout naturellement imposées pour des questions de sécurité : `LD_PRELOAD` est ignoré par les programmes en suid :).

Le mécanisme de détournement implémente la notion de « surcharge » connue en C++. La `libdl.so` permet aussi d'appeler les fonctions surchargées à l'intérieur des surcharges.

6. Chaînage de fonctions : de l'héritage aux chaînes de traitements

Supposons que nous ayons en mémoire plusieurs bibliothèques qui possèdent des points d'entrée portant le même nom. En généralisant le processus de surcharge, la fonction `dlsym` permet de chaîner ces fonctions.

```
% for i in 1 2 3 4
% do
% cat > lib$i.c <<EOF
lib1.c:
#include <stdio.h>
#include <dlfcn.h>
void foo() {
    void (*next_foo)(void);
    printf("lib$.foo()\n");
    next_foo = dlsym(RTLF_NEXT, "foo");
    next_foo();
}
EOF
% gcc -shared -fPIC -G lib$i.c -o lib$i.so
% done
% cat > foo.c <<EOF
#include <dlfcn.h>
extern void foo();
int main() {
    foo();
}
EOF
% gcc foo.c -o foo -L. -l1 -l2 -l3 -l4 -ldl
```

Pour cet exemple, il est nécessaire que la variable `LD_LIBRARY_PATH` contienne le répertoire courant. Le lancement de `foo` donne :

```
% foo
lib1.foo()
lib2.foo()
lib3.foo()
lib4.foo()
```

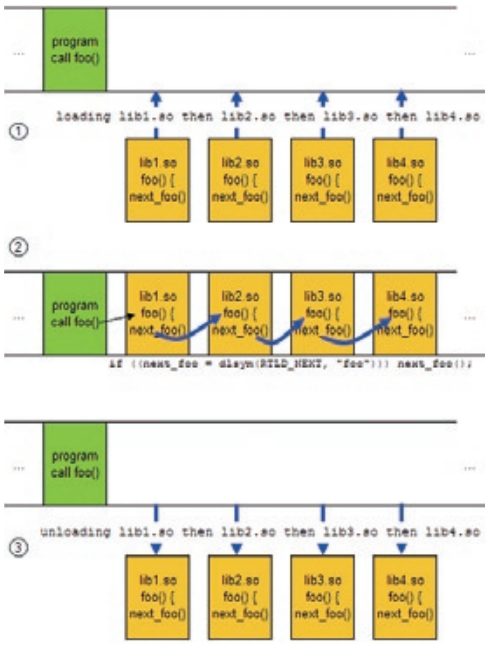
L'ordre d'appel dépend de l'ordre d'inclusion des bibliothèques :

```
% gcc foo.c -o foo -L. -l3 -l2 -l1 -l4 -ldl
% foo
lib3.foo()
lib2.foo()
lib1.foo()
lib4.foo()
```

Voyons s'il en va de même si les bibliothèques sont chargées dynamiquement avec `dlopen`. Il est nécessaire dans ce cas de préciser à l'éditeur de liens à la volée de résoudre tous les points d'entrée dès le chargement et de rendre les symboles globaux (figure 3).

On remplacera dont l'argument `RTLD_LAZY` de `dlopen` par l'argument `RTLD_NOW|RTLD_GLOBAL`.

Figure 3 : Chaînage dynamique de fonctions



(1) Le programme charge dynamiquement les objets partagés `lib1.so`, `lib2.so`, `lib3.so`, puis `lib4.so`.

(2) Le programme appelle la fonction `foo()` déclarée dans la première des bibliothèques chargées. La fonction `foo()` est appelée, effectue son traitement, puis cherche à résoudre le symbole `foo` dans la bibliothèque suivante.

Si ce symbole existe, elle appelle la fonction...

(3) Le programme décharge les bibliothèques.

```
% for i in 1 2 3 4
% cat > libchain$i.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
void foo() {
   void (*next_foo)();
   printf("libchain$i.foo()\n");
   if ((next_foo=(void(*)())dlsym(RTLD_NEXT,"foo")))
        next_foo();
}
EOF
% gcc -shared -fPIC -G libchain$i.c -o libchain$i.so -ldl
% done
% cat > chain.c <<EOF
#include <dlfcn.h>
int main() {
   void *l1, *l2, *l3, *l4;
   void (*foo)();
   l1 = dlopen("libchain1.so", RTLD_NOW | RTLD_GLOBAL);
   l2 = dlopen("libchain2.so", RTLD_NOW | RTLD_GLOBAL);
   l3 = dlopen("libchain3.so", RTLD_NOW | RTLD_GLOBAL);
   l4 = dlopen("libchain4.so", RTLD_NOW | RTLD_GLOBAL);
   foo = (void (*)()) dlsym(l1, "foo");
   foo();
   dlclose(l4);
   dlclose(l3);
   dlclose(l2);
   dlclose(l1);
   return 0;
}
EOF
% gcc chain.c -o chain -ldl
```

Le lancement de `chain` sous Solaris nous donnera :

```
% chain
libchain1.foo()
libchain2.foo()
libchain3.foo()
libchain4.foo()
```

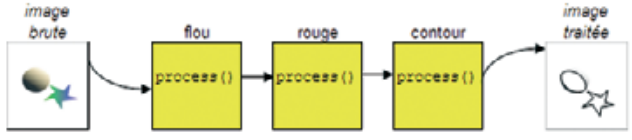
Par contre, sous Linux, `chain` ne nous donnera que :

```
% chain
libchain1.foo()
```

Dans POSIX **[3]**, `RTLD_NEXT` est officiellement « réservé pour une utilisation future » (partie normative). On trouve cependant une section informative (i. e. non normative), qui explique le fonctionnement que `RTLD_NEXT` devra avoir. La LSB **[4]** (depuis au moins la version 1.3) spécifie que « La valeur `RTLD_NEXT`, qui est réservée pour une utilisation future, devrait être disponible, avec le comportement décrit dans ISO POSIX (2003) ». La plupart des pages de manuels sont aussi en adéquation avec la section informative. En conséquence, un bug a été déposé dans le BUGZILLA **[8]** de la glibc. Cependant, eu égard aux positions assez radicales prises par Ulrich Drepper, responsable de la glibc, au sujet de la LSB (« *Do you still think the LSB has some value ?* »), le bug a été, après quelques péripéties, considéré comme invalide par M. Drepper lui-même. Les linuxiens devront donc se rabattre sur la `libtld.so` pour tirer pleinement parti de cette fonctionnalité.

D'ailleurs, que pourrait-on bien en faire ? Le chaînage de fonction permet la création de chaînes de traitement, à l'instar d'une chaîne de montage industrielle. Ce chaînage permet la composition de processus spécialisés. Imaginons, par exemple, que nous fassions du traitement d'image. Une image donnée nécessite une série d'opérations (filtres). Supposons que nous disposions d'une dizaine de filtres spécifiques (flou, rouge, vert, bleu...). Nous pouvons dynamiquement définir un ordre de traitement de notre image de base, puis, par sauts d'une bibliothèque à l'autre.

Figure 4 : Chaîne de traitement



7. Espionnage de greffons

Lorsqu'on programme des greffons, il arrive que ces greffons ne se chargent pas correctement, que certains symboles ne soient pas définis, ou encore que le greffon chargé ne soit pas celui attendu (problème classique du polymorphisme). Il arrive de plus que le programme greffé ne nous transmette pas les messages d'erreur de la `libdl.so`. Dans ces cas, il est assez difficile de trouver les causes des dysfonctionnements.

Pour contourner le problème, on peut détourner les fonctions `dlsym`, `dlopen`, `dlerror` et `dlclose`. Oui, mais voilà, en détournant `dlsym`, comment récupérer le point d'entrée `dlsym` du système ?

Certains systèmes nous facilitent la tâche. Un `nm` de `/usr/lib/libdl.so.1` sur Solaris 8 nous montre que les symboles `dlsym`, `dlopen`, `dlclose` et `dlerror` ont des contreparties en `_dlsym`, `_dlopen`, `_dlclose` et `_dlerror` :

```
nm /usr/lib/libdl.so.1 | egrep "dlsym|dlopen|dlclose|dlerror"
[37]   |  2236|   8|FUNC |GLOB |0    |7      |_dlclose
[35]   |  2244|   8|FUNC |GLOB |0    |7      |_dlerror
[27]   |  2220|   8|FUNC |GLOB |0    |7      |_dlopen
[55]   |  2228|   8|FUNC |GLOB |0    |7      |_dlsym
[26]   |  2236|   8|FUNC |WEAK |0    |7      |dlclose
[53]   |  2244|   8|FUNC |WEAK |0    |7      |dlerror
[47]   |  2220|   8|FUNC |WEAK |0    |7      |dlopen
[44]   |  2228|   8|FUNC |WEAK |0    |7      |dlsym
```

Les points d'entrée standard semblent bien être des *alias* de plus faible priorité que les originaux préfixés par un souligné. Dans ce cas, il est possible d'appeler directement `_dlsym` à l'intérieur de notre fonction détournée.

Sous Linux, par contre, ces alias n'existent pas. Un `nm` de la `libdl.so` ne nous apprendra en fait rien dans les distributions où les bibliothèques système sont débarrassées de leurs symboles (`strip`). L'utilitaire `string` nous donnera les points d'entrée. Las, aucun de ces points d'entrée, après consultation des sources de la glibc (**http://sources.redhat.com/cgi-bin/cvsweb.cgi/libc/dlfcn/?cvsroot=glibc#dirlist**) et quelques tests ne nous sera d'un grand secours.

Il faut alors se tourner vers la fonction `dlvsym(void *handle, const char *name, const char *version_str)` dont la documentation laisse encore à désirer, quelle qu'en soit la source. Cette fonction nécessite, en plus du nom du point d'entrée, une chaîne de caractères représentant la version. Pour retrouver cette chaîne, il suffit d'écrire un petit bout de code faisant appel à `dlsym`, de le compiler et le lier avec la `libdl.so`, puis de faire un `nm` sur le binaire.

```
% nm dltest
...
08049840 W data_start
         U dlclose@@GLIBC_2.0
         U dlopen@@GLIBC_2.1
         U dlsym@@GLIBC_2.0
...
```

Cette chaîne est accolée au symbole `dlsym` sous la forme `dlsym@@version_st` :

```
% nm dltest | grep dlsym | cut -d@ -f3
GLIBC_2.0
```

Une fois cette chaîne trouvée, nous pouvons résoudre le point d'entré, et réécrire nos fonctions :

```
#include <stdio.h>
#include <stdarg.>
#ifdef  _LINUX_
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#endif
#include <dlfcn.h>

typedef struct {
    void *handle;
    void *(*open)(const char  *, int);
    void *(*sym)(void *, const char *);
    char *(*error)(void);
    int   (*close)(void *);
} dlspy_t;

dlspy_t __dlspy__ = { NULL, NULL, NULL, NULL, NULL };

#ifdef LINUX
#include "config.h"
#ifndef DLSPY_GLIBC_VERSION
#define DLSPY_GLIBC_VERSION "GLIBC_2.0"
#endif
#endif
void
dlspy_init() {
    /* 1st step: save standard dl* functions */
#if defined(SUN)
    __dlspy__.sym   =
       (void *(*)(void *, const char *)) _dlsym;
    __dlspy__.open  =
       (void *(*)(const char *, int))    _dlopen;
    __dlspy__.error =
       (char *(*)(void))                 _dlerror;
    __dlspy__.close =
       (int   (*)(void *))               _dlclose;
#elif defined(LINUX)
    if (!(__dlspy__.sym = dlvsym(RTLD_NEXT,
          "dlsym", DLSPY_GLIBC_VERSION))) {
       perror("dlspy fatal: cannot fetch "
         "system's dlsym entry point\n");
       exit(1);
    }
   __dlspy__.open  = (void *(*)(const char *, int))
      __dlspy__.sym(RTLD_NEXT, "dlopen");
    __dlspy__.error = (char *(*)(void))
      __dlspy__.sym(RTLD_NEXT, "dlerror");
    __dlspy__.close = (int   (*)(void *))
      __dlspy__.sym(RTLD_NEXT, "dlclose");
#else
#error ERROR: cannot get dl* original entry points.;
#endif
}
void
dlspy_echo(char *fmt, ...) {
    va_list args;
    char buf[1024], buf2[1024];
    va_start(args, fmt);
    sprintf(buf2, "dlspy: %s", fmt);
    vsprintf(buf, buf2, args);
    fprintf(stderr, "%s\n", buf);
    va_end(args);
}
char *

dlerror(void) {
    static char *errstr = NULL;
    char *t;
    if (!__dlspy__.open) dlspy_init();
    t = __dlspy__.error();
    if (!t && errstr) return errstr;
    if (t) {
        errstr = t;
        return t;
    }
    return (char *) strdup(" ");
}
void *
dlopen(const char *name, int mode) {
    void *t = NULL;
    if (!__dlspy__.open) dlspy_init();
```

```
    dlspy_echo("try to open \"%s\" shared object"
      " with mode %d...", name, mode);
    if (!(t = __dlspy__.open(name, mode)))
      dlspy_echo("failed (%s)\n", dlerror());
    else dlspy_echo("ok\n");
    return t;
}
void *
dlsym(void *h, const char *name) {
    void *t = NULL;
    if (!__dlspy__.open) dlspy_init();
    dlspy_echo("try to bind symbol \"%s\" "
      "as new entry point...", name);
    if (!(t = __dlspy__.sym(h, name)))
      dlspy_echo("failed (%s)\n", dlerror());
    else dlspy_echo("ok at %x\n", t);
    return t;
}
int
dlclose(void *h) {
    if (!__dlspy__.open) dlspy_init();
    dlspy_echo("dlclose(%p)", h);
    return __dlspy__.close(h);
}
```

Peut-on aller plus loin ? Une fonctionnalité intéressante serait de pouvoir tracer chaque appel aux fonctions résolues par `dlsym()`. Pour ce faire, il semble nécessaire de définir un pool de fonctions qui devront, lors du premier appel, prendre l'adresse du point d'entrée fraîchement résolu, et, dans un second temps, appeler le point d'entrée avec les paramètres qui lui ont été passés, sans pour autant connaître leur nombre ni leur taille.

Ce problème n'est pas nouveau. La FAQ du groupe de discussion **comp.lang.c** **[9]** en parle comme d'un problème insoluble de manière générique. Des pointeurs ou axes de recherches y sont explorés. On peut le résoudre en grande partie par l'utilisation de certaines fonctionnalités obscures de gcc **[10]**, les `__builtin_apply` et `__builtin_apply_args`.

8. Et en C++ ?

Il est possible d'utiliser la `libdl.so` en C++. Cependant, il est nécessaire de faire un peu plus attention qu'en C. En effet, l'implémentation du polymorphisme (mais qui pourrait tout autant servir à vérifier le typage en cours d'exécution) en C++ passe par l'ajout de préfixes et de suffixes aux noms de fonction. C'est la technique du *mangling*. Aussi, est-il nécessaire, comme pour l'utilisation de pointeurs de fonctions, de définir les fonctions exportées en « `extern "C"` » afin d'éviter le mangling.

Pour aller plus loin sur le sujet, voir les documents cités en référence **[11]** et **[12]**.

Conclusion

Cette présentation de la `libdl` est loin d'être exhaustive. Les possibilités qu'elle offre, comme la surcharge ou l'héritage à un niveau très différent d'un langage de programmation en font un outil très puissant dont la seule limite est celle notre propre imagination.

Yann Langlais,

RÉFÉRENCES

- **[1]** *Executable and Linkable Format* : **http://en.wikipedia.org/wiki/Executable_and_Linkable_Format**
  Tool Interface Standard (TIS) : Executable and Linkable Format (ELF) Specification v1.2 : **http://refspecs.freestandards.org/elf/elf.pdf**
  Tool Interface Standard (TIS) : Generic ELF Specification ELFVERSION : **http://www.linuxbase.org/spec/book/ELF-generic/ELF-generic.html**
  LEVINE (John), *Linkers and Loaders*, *Chapter 10* : « *Dynamic Linking and Loading* », **http://www.iecc.com/linker/linker10.html**
- **[2]** *The GNU Libtool Homepage* : **http://www.gnu.org/software/libtool/** **http://www.gnu.org/software/libtool/manual.html**
- **[3]** *The Open Group Base Specifications Issue 6 IEEE Std 1003.1*, 2004 Edition (i. e. la norme POSIX)
  Single Unix Specification : **http://www.unix.org/single_unix_specification/**
  Voir les spécifications de dlopen : **http://www.opengroup.org/onlinepubs/009695399/functions/dlsym.html**
  Ce site demande un enregistrement gratuit préalable.
- **[4]** *Linux Standard Base 3.0*
  13.16. Interface Definitions for libdl : **http://refspecs.freestandards.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/libdlman.html**
  13.15. Data Definitions for libdl : **http://refspecs.freestandards.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/libdl-ddefs.html**
  dlsym : **http://refspecs.freestandards.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/baselib-dlsym-1.html**
- **[5]** *Linux Programer's Manual* : **http://www.frech.ch/man/man3/dlopen.3.html**
- **[6]** *Linux HOWTO* : **http://www.faqs.org/docs/Linux-HOWTO/Program-Library-HOWTO.html**
  How to Write Shared Libraries : http://people.redhat.com/~drepper/dsohowto.pdf
- **[7]** L'éditeur de lien et ses variables d'environnement : DRALET (Samuel), *GNU/Linux Magazine France* numéro 63, juillet/août 2004, page 76.
- **[8]** Descriptif et historique du bug 1319 sur le comportement de dlsym avec RTLD_NEXT : **http://sourceware.org/bugzilla/show_bug.cgi?id=1319**
- **[9]** *comp.lang.c Frequently Asked Questions : 15. Variable-Length Argument Lists* : **http://www.eskimo.com/~scs/C-faq/s15.html**
- **[10]** *gcc info : Construction calls* : **http://www.cs.cmu.edu/cgi-bin/info2www?(gcc.info)Constructing%2520Calls**
  gcc features : WOLF (Clifford), « *Some demonstrations of nice/osbcure gcc features* », **http://www.clifford.at/cfun/gccfeat/**
- **[11]** *C++ dlopen mini HOWTO* : **http://www.isotton.com/howtos/C++-dlopen-mini-HOWTO/C++-dlopen-mini-HOWTO.html**
- **[12]** NORTON (James), « *Dynamic Class Loading for C++ on Linux* », *Linux Journal*, **http://www.linuxjournal.com/article/3687**
  LANGLAIS (Yann) : **http://ilay.org/yann/articles/**

Creative Commons

Paternité - Pas d'Utilisation Commerciale - Pas de Modification 2.0

Creative Commons n'est pas un cabinet d'avocats et ne fournit pas de services de conseil juridique. La distribution de la présente version de ce contrat ne crée aucune relation juridique entre les parties au contrat présenté ci-après et Creative Commons. Creative Commons fournit cette offre de contrat-type en l'état, à seule fin d'information. Creative Commons ne saurait être tenu responsable des éventuels préjudices résultant du contenu ou de l'utilisation de ce contrat.

Contrat

L'Oeuvre (telle que définie ci-dessous) est mise à disposition selon les termes du présent contrat appelé Contrat Public Creative Commons (dénommé ici « CPCC » ou « Contrat »). L'Oeuvre est protégée par le droit de la propriété littéraire et artistique (droit d'auteur, droits voisins, droits des producteurs de bases de données) ou toute autre loi applicable. Toute utilisation de l'Oeuvre autrement qu'explicitement autorisée selon ce Contrat ou le droit applicable est interdite.

L'exercice sur l'Oeuvre de tout droit proposé par le présent contrat vaut acceptation de celui-ci. Selon les termes et les obligations du présent contrat, la partie Offrante propose à la partie Acceptante l'exercice de certains droits présentés ci-après, et l'Acceptant en approuve les termes et conditions d'utilisation.

1. Définitions

a. **« Oeuvre »** : oeuvre de l'esprit protégeable par le droit de la propriété littéraire et artistique ou toute loi applicable et qui est mise à disposition selon les termes du présent Contrat.
b. **« Oeuvre dite Collective »** : une oeuvre dans laquelle l'oeuvre, dans sa forme intégrale et non modifiée, est assemblée en un ensemble collectif avec d'autres contributions qui constituent en elles-mêmes des oeuvres séparées et indépendantes. Constituent notamment des Oeuvres dites Collectives les publications périodiques, les anthologies ou les encyclopédies. Aux termes de la présente autorisation, une oeuvre qui constitue une Oeuvre dite Collective ne sera pas considérée comme une Oeuvre dite Dérivée (telle que définie ci-après).
c. **« Oeuvre dite Dérivée »** : une oeuvre créée soit à partir de l'Oeuvre seule, soit à partir de l'Oeuvre et d'autres oeuvres préexistantes. Constituent notamment des Oeuvres dites Dérivées les traductions, les arrangements musicaux, les adaptations théâtrales, littéraires ou cinématographiques, les enregistrements sonores, les reproductions par un art ou un procédé quelconque, les résumés, ou toute autre forme sous laquelle l'Oeuvre puisse être remaniée, modifiée, transformée ou adaptée, à l'exception d'une oeuvre qui constitue une Oeuvre dite Collective. Une Oeuvre dite Collective ne sera pas considérée comme une Oeuvre dite Dérivée aux termes du présent Contrat. Dans le cas où l'Oeuvre serait une composition musicale ou un enregistrement sonore, la synchronisation de l'oeuvre avec une image animée sera considérée comme une Oeuvre dite Dérivée pour les propos de ce Contrat.
d. **« Auteur original »** : la ou les personnes physiques qui ont créé l'Oeuvre.
e. **« Offrant »** : la ou les personne(s) physique(s) ou morale(s) qui proposent la mise à disposition de l'Oeuvre selon les termes du présent Contrat.
f. **« Acceptant »** : la personne physique ou morale qui accepte le présent contrat et exerce des droits sans en avoir violé les termes au préalable ou qui a reçu l'autorisation expresse de l'Offrant d'exercer des droits dans le cadre du présent contrat malgré une précédente violation de ce contrat.

2. Exceptions aux droits exclusifs. Aucune disposition de ce contrat n'a pour intention de réduire, limiter ou restreindre les prérogatives issues des exceptions aux droits, de l'épuisement des droits ou d'autres limitations aux droits exclusifs des ayants droit selon le droit de la propriété littéraire et artistique ou les autres lois applicables.

3. Autorisation. Soumis aux termes et conditions définis dans cette autorisation, et ceci pendant toute la durée de protection de l'Oeuvre par le droit de la propriété littéraire et artistique ou le droit applicable, l'Offrant accorde à l'Acceptant l'autorisation mondiale d'exercer à titre gratuit et non exclusif les droits suivants :

a. reproduire l'Oeuvre, incorporer l'Oeuvre dans une ou plusieurs Oeuvres dites Collectives et reproduire l'Oeuvre telle qu'incorporée dans lesdites Oeuvres dites Collectives;
b. distribuer des exemplaires ou enregistrements, présenter, représenter ou communiquer l'Oeuvre au public par tout procédé technique, y compris incorporée dans des Oeuvres Collectives;
c. lorsque l'Oeuvre est une base de données, extraire et réutiliser des parties substantielles de l'Oeuvre.

Les droits mentionnés ci-dessus peuvent être exercés sur tous les supports, médias, procédés techniques et formats. Les droits ci-dessus incluent le droit d'effectuer les modifications nécessaires techniquement à l'exercice des droits dans d'autres formats et procédés techniques. L'exercice de tous les droits qui ne sont pas expressément autorisés par l'Offrant ou dont il n'aurait pas la gestion demeure réservé, notamment les mécanismes de gestion collective obligatoire applicables décrits à l'article 4(d).

4. Restrictions. L'autorisation accordée par l'article 3 est expressément assujettie et limitée par le respect des restrictions suivantes :

a. L'Acceptant peut reproduire, distribuer, représenter ou communiquer au public l'Oeuvre y compris par voie numérique uniquement selon les termes de ce Contrat. L'Acceptant doit inclure une copie ou l'adresse Internet (Identifiant Uniforme de Ressource) du présent Contrat à toute reproduction ou enregistrement de l'Oeuvre que l'Acceptant distribue, représente ou communique au public y compris par voie numérique. L'Acceptant ne peut pas offrir ou imposer de conditions d'utilisation de l'Oeuvre qui altèrent ou restreignent les termes du présent Contrat ou l'exercice des droits qui y sont accordés au bénéficiaire. L'Acceptant ne peut pas céder de droits sur l'Oeuvre. L'Acceptant doit conserver intactes toutes les informations qui renvoient à ce Contrat et à l'exonération de responsabilité. L'Acceptant ne peut pas reproduire, distribuer, représenter ou communiquer au public l'Oeuvre, y compris par voie numérique, en utilisant une mesure technique de contrôle d'accès ou de contrôle d'utilisation qui serait contradictoire avec les termes de cet Accord contractuel. Les mentions ci-dessus s'appliquent à l'Oeuvre telle qu'incorporée dans une Oeuvre dite Collective, mais, en dehors de l'Oeuvre en elle-même, ne soumettent pas l'Oeuvre dite Collective, aux termes du présent Contrat. Si l'Acceptant crée une Oeuvre dite Collective, à la demande de tout Offrant, il devra, dans la mesure du possible, retirer de l'Oeuvre dite Collective toute référence au dit Offrant, comme demandé. Si l'Acceptant crée une Oeuvre dite Collective, à la demande de tout Auteur, il devra, dans la mesure du possible, retirer de l'Oeuvre dite Collective toute référence au dit Auteur, comme demandé.

b. L'Acceptant ne peut exercer aucun des droits conférés par l'article 3 avec l'intention ou l'objectif d'obtenir un profit commercial ou une compensation financière personnelle. L'échange de l'Oeuvre avec d'autres Oeuvres protégées par le droit de la propriété littéraire et artistique par le partage électronique de fichiers, ou par tout autre moyen, n'est pas considéré comme un échange avec l'intention ou l'objectif d'un profit commercial ou d'une compensation financière personnelle, dans la mesure où aucun paiement ou compensation financière n'intervient en relation avec l'échange d'Oeuvres protégées.
c. Si l'Acceptant reproduit, distribue, représente ou communique l'Oeuvre au public, y compris par voie numérique, il doit conserver intactes toutes les informations sur le régime des droits et en attribuer la paternité à l'Auteur Original, de manière raisonnable au regard au médium ou au moyen utilisé. Il doit communiquer le nom de l'Auteur Original ou son éventuel pseudonyme s'il est indiqué ; le titre de l'Oeuvre Originale s'il est indiqué ; dans la mesure du possible, l'adresse Internet ou Identifiant Uniforme de Ressource (URI), s'il existe, spécifié par l'Offrant comme associé à l'Oeuvre, à moins que cette adresse ne renvoie pas aux informations légales (paternité et conditions d'utilisation de l'Oeuvre). Ces obligations d'attribution de paternité doivent être exécutées de manière raisonnable. Cependant, dans le cas d'une Oeuvre dite Collective, ces informations doivent, au minimum, apparaître à la place et de manière aussi visible que celles à laquelle apparaissent les informations de même nature.
d. Dans le cas où une utilisation de l'Oeuvre serait soumise à un régime légal de gestion collective obligatoire, l'Offrant se réserve le droit exclusif de collecter ces redevances par l'intermédiaire de la société de perception et de répartition des droits compétente. Sont notamment concernés la radiodiffusion et la communication dans un lieu public de phonogrammes publiés à des fins de commerce, certains cas de retransmission par câble et satellite, la copie privée d'Oeuvres fixées sur phonogrammes ou vidéogrammes, la reproduction par reprographie.

5. Garantie et exonération de responsabilité

a. En mettant l'Oeuvre à la disposition du public selon les termes de ce Contrat, l'Offrant déclare de bonne foi qu'à sa connaissance et dans les limites d'une enquête raisonnable :
    i. L'Offrant a obtenu tous les droits sur l'Oeuvre nécessaires pour pouvoir autoriser l'exercice des droits accordés par le présent Contrat, et permettre la jouissance paisible et l'exercice licite de ces droits, ceci sans que l'Acceptant n'ait aucune obligation de verser de rémunération ou tout autre paiement ou droits, dans la limite des mécanismes de gestion collective obligatoire applicables décrits à l'article 4(e);
b. L'Oeuvre n'est constitutive ni d'une violation des droits de tiers, notamment du droit de la propriété littéraire et artistique, du droit des marques, du droit de l'information, du droit civil ou de tout autre droit, ni de diffamation, de violation de la vie privée ou de tout autre préjudice délictuel à l'égard de toute tierce partie.
c. A l'exception des situations expressément mentionnées dans le présent Contrat ou dans un autre accord écrit, ou exigées par la loi applicable, l'Oeuvre est mise à disposition en l'état sans garantie d'aucune sorte, qu'elle soit expresse ou tacite, y compris à l'égard du contenu ou de l'exactitude de l'Oeuvre.

6. Limitation de responsabilité. A l'exception des garanties d'ordre public imposées par la loi applicable et des réparations imposées par le régime de la responsabilité vis-à-vis d'un tiers en raison de la violation des garanties prévues par l'article 5 du présent contrat, l'Offrant ne sera en aucun cas tenu responsable vis-à-vis de l'Acceptant, sur la base d'aucune théorie légale ni en raison d'aucun préjudice direct, indirect, matériel ou moral, résultant de l'exécution du présent Contrat ou de l'utilisation de l'Oeuvre, y compris dans l'hypothèse où l'Offrant avait connaissance de la possible existence d'un tel préjudice.

7. Résiliation

a. Tout manquement aux termes du contrat par l'Acceptant entraîne la résiliation automatique du Contrat et la fin des droits qui en découlent. Cependant, le contrat conserve ses effets envers les personnes physiques ou morales qui ont reçu de la part de l'Acceptant, en exécution du présent contrat, la mise à disposition d'Oeuvres dites Dérivées, ou d'Oeuvres dites Collectives, ceci tant qu'elles respectent pleinement leurs obligations. Les sections 1, 2, 5, 6 et 7 du contrat continuent à s'appliquer après la résiliation de celui-ci.
b. Dans les limites indiquées ci-dessus, le présent Contrat s'applique pendant toute la durée de protection de l'Oeuvre selon le droit applicable. Néanmoins, l'Offrant se réserve à tout moment le droit d'exploiter l'Oeuvre sous des conditions contractuelles différentes, ou d'en cesser la diffusion; cependant, le recours à cette option ne doit pas conduire à retirer les effets du présent Contrat (ou de tout contrat qui a été ou doit être accordé selon les termes de ce Contrat), et ce Contrat continuera à s'appliquer dans tous ses effets jusqu'à ce que sa résiliation intervienne dans les conditions décrites ci-dessus.

8. Divers

a. A chaque reproduction ou communication au public par voie numérique de l'Oeuvre ou d'une Oeuvre dite Collective par l'Acceptant, l'Offrant propose au bénéficiaire une offre de mise à disposition de l'Oeuvre dans des termes et conditions identiques à ceux accordés à la partie Acceptante dans le présent Contrat.
b. La nullité ou l'inapplicabilité d'une quelconque disposition de ce Contrat au regard de la loi applicable n'affecte pas celle des autres dispositions qui resteront pleinement valides et applicables. Sans action additionnelle par les parties à cet accord, lesdites dispositions devront être interprétées dans la mesure minimum nécessaire à leur validité et leur applicabilité.
c. Aucune limite, renonciation ou modification des termes ou dispositions du présent Contrat ne pourra être acceptée sans le consentement écrit et signé de la partie compétente.
d. Ce Contrat constitue le seul accord entre les parties à propos de l'Oeuvre mise ici à disposition. Il n'existe aucun élément annexe, accord supplémentaire ou mandat portant sur cette Oeuvre en dehors des éléments mentionnés ici. L'Offrant ne sera tenu par aucune disposition supplémentaire qui pourrait apparaître dans une quelconque communication en provenance de l'Acceptant. Ce Contrat ne peut être modifié sans l'accord mutuel écrit de l'Offrant et de l'Acceptant.
e. Le droit applicable est le droit français.

Creative Commons n'est pas partie à ce Contrat et n'offre aucune forme de garantie relative à l'Oeuvre. Creative Commons décline toute responsabilité à l'égard de l'Acceptant ou de toute autre partie, quel que soit le fondement légal de cette responsabilité et quel que soit le préjudice subi, direct, indirect, matériel ou moral, qui surviendrait en rapport avec le présent Contrat. Cependant, si Creative Commons s'est expressément identifié comme Offrant pour mettre une Oeuvre à disposition selon les termes de ce Contrat, Creative Commons jouira de tous les droits et obligations d'un Offrant.

A l'exception des fins limitées à informer le public que l'Oeuvre est mise à disposition sous CPCC, aucune des parties n'utilisera la marque « Creative Commons » ou toute autre indication ou logo afférent sans le consentement préalable écrit de Creative Commons. Toute utilisation autorisée devra être effectuée en conformité avec les lignes directrices de Creative Commons à jour au moment de l'utilisation, telles qu'elles sont disponibles sur son site Internet ou sur simple demande.

Creative Commons peut être contacté à http://creativecommons.org/.