

# Reverse Engineering avec LD\_PRELOAD

<http://security.nnov.ru/articles/reveng/>

Ce document traite la fonctionnalité LD\_PRELOAD, et comment elle peut être utile pour le reverse engineering d'exécutables liés dynamiquement.

Cette technique vous permet de duper des fonctions/injecter du code et manipuler le flux de l'application.

## Méthodes de Compilation

En général, il y a deux manières de produire un exécutable.

La première méthode est la **compilation statique**. Pendant le processus de compilation, l'exécutable qui va être produit, va être indépendant et inclure tout ce dont il a besoin pour fonctionner correctement. L'avantage de cette méthode est principalement la portabilité, du fait que l'utilisateur n'est pas obligé d'installer quelque chose pour utiliser l'application. Le désavantage est un exécutable relativement démesurément gros, et les bugs qui proviennent d'un composant externe ne seront pas corrigés, tant que l'exécutable n'est pas linké.

L'autre méthode est l'**exécutable produit dynamiquement**. Il est dépendant aux bibliothèques partagées pour fonctionner, et correspond aux changements dans les bibliothèques partagées. Pour le meilleur ou pour le pire, cela peut être un avantage et un désavantage. Les exécutables linkés dynamiquement, de part leur nature, sont plus légers que les exécutables produits statiquement.

Le compilateur gcc produira par défaut un exécutable lié dynamiquement, à moins que vous lui spécifiez le paramètre ‘-static’.

Vous pouvez naviguer dans la liste des dépendances de l'exécutable lié dynamiquement en utilisant l'utilitaire ldd.

Exemple :

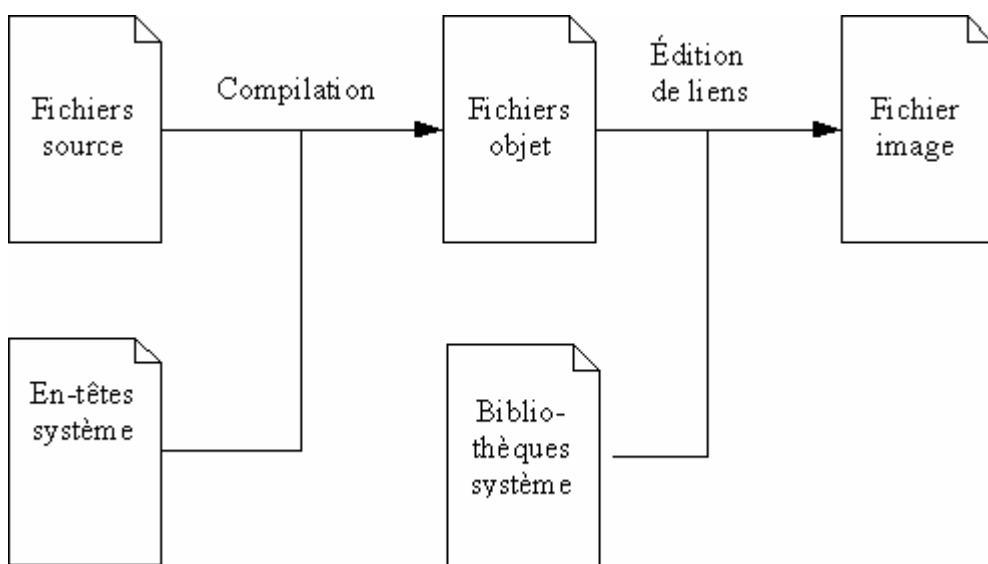
```
root@magicbox:~# ldd /bin/ls
 linux-gate.so.1 => (0xfffffe000)
 librt.so.1 => /lib/tls/librt.so.1 (0xb7fd7000)
 libc.so.6 => /lib/tls/libc.so.6 (0xb7eba000)
 libpthread.so.0 => /lib/tls/libpthread.so.0 (0xb7ea8000)
 /lib/ld-linux.so.2 (0xb7feb000)
root@magicbox:~#
```

Chaque dépendance présente la librairie dont elle dépend, et l'adresse que nous voyons, est celle vers laquelle elle sera mappée à l'exécution. Cet article se rapporte aux exécutables liés dynamiquement car ils peuvent être manipulés par LD\_PRELOAD.

## **Editeur de liens à l'exécution (Runtime linker)**

Ce programme regarde dans tous les fichiers objets les références partielles aux autres fichiers objets et, pour chaque « lien » ainsi trouvé, il complète les informations nécessaires pour en faire une référence complète. Par exemple, un fichier source peut très bien utiliser une fonctionnalité d'un autre fichier source. Comme cette fonctionnalité n'est pas définie dans le fichier source courant, une référence partielle est créée dans le fichier objet lors de sa compilation, mais il faudra tout de même la terminer en indiquant exactement comment accéder à la fonctionnalité externe. C'est le travail de l'éditeur de liens, lorsqu'il regroupe les deux fichiers objets.

Certains fichiers objets sont nécessaires pour tous les programmes. Ce sont notamment les fichiers objets définissant les fonctions de base, et les fonctions permettant d'accéder au système d'exploitation. Ces fichiers objets ont donc été regroupés dans des bibliothèques (également couramment appelées « librairies »), que l'on peut ainsi utiliser directement lors de la phase d'édition de liens. Les fichiers objets nécessaires sont alors lus dans la bibliothèque et ajoutés au programme en cours d'édition de liens. Les bibliothèques portent souvent l'extension « .a » (ou « .lib » dans les systèmes Microsoft).



LD\_PRELOAD est une variable d'environnement qui affecte l'éditeur de liens. Elle permet de mettre en place un objet dynamique, qui va créer une espèce de couche du tampon, entre les références de l'application et les dépendances. Il donne aussi la possibilité de faire un lien entre l'application et la réallocation des symboles/références.

Pour simplifier la situation, il s'agit d'une attaque man-in-the middle entre le programme et les bibliothèques nécessaires ;)

### **Limitations**

L'option LD\_PRELOAD n'affecte pas les applications qui ont été chmodées avec le drapeau +s (setgid/setuid). A moins que l'utilisateur qui exécute le programme possède déjà les priviléges root. De même que sur certaines plateformes, pour pouvoir utiliser l'option LD\_PRELOAD, l'utilisateur doit déjà posséder les priviléges root. Tous les exemples donnés dans ce document considèrent que vous utilisez votre propre machine. Mais cela fonctionnera de la même manière, si vous possédez les priviléges root ailleurs.

## Mise en pratique

La chose la plus simple que permet LD\_PRELOAD est de duper une fonction. Dans cette optique, nous allons créer une bibliothèque partagée qui va inclure l'implémentation de la fonction que nous souhaitons dupler. Mais avant cela, jetons un œil à notre challenge :

```
/*
 * strcmp-target.c, Un challenge simple qui compare deux chaines
 */
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {

    char passwd[] = "foobar";

    if (argc < 2) {
        printf("usage: %s <MotDePasse>\n", argv[0]);
        return 0;
    }

    if (!strcmp(passwd, argv[1])) {
        printf("Feu vert!\n");
        return 1;
    }

    printf("Feu rouge!\n");
    return 0;
}
```

Dans cet exemple, nous avons un simple challenge de comparaison de mot de passe. Il utilise la fonction strcmp() pour comparer les deux chaînes. Attaquons-le en duplant la fonction strcmp() pour voir qui va être exécuté et pour nous assurer que cela va toujours retourner des chaînes identiques.

```

#include <stdio.h>
#include <string.h>

/*
 * strcmp-hijack.c, Usurpe la fonction strcmp()
 */

/*
 * strcmp, Fonction strcmp fixée – Toujours égal!
 */

int strcmp(const char *s1, const char *s2) {

printf("S1 eq %s\n", s1);
printf("S2 eq %s\n", s2);

// RETOURNE TOUJOURS DES CHAINES IDENTIQUES!
return 0;
}

```

Maintenant compilons notre strcmp-hijack.c en tant que bibliothèque partagée, en faisant:

```

root@magicbox:/tmp# gcc -fPIC -c strcmp-hijack.c -o strcmp-hijack.o
root@magicbox:/tmp# gcc -shared -o strcmp-hijack.so strcmp-hijack.o

```

Avant d'attaquer, vérifions que notre challenge fait bien ce que nous attendons, en faisant:

```

root@magicbox:/tmp# ./strcmp-target redbull
Feu rouge!
root@magicbox:/tmp#

```

Maintenant attaquons-le en utilisant LD\_PRELOAD, en faisant:

```

root@magicbox:/tmp# LD_PRELOAD="./strcmp-hijack.so" ./strcmp-target redbull
S1 eq foobar
S2 eq redbull
Feu vert!
root@magicbox:/tmp#

```

Notre bibliothèque partagée a fait son travail. Nous avons usurper la fonction strcmp() et l'avons fait retourner une valeur prédéfinie. Nous mettons aussi un printf de débogage qui affiche la valeur des deux arguments qui ont été envoyés à la fonction.

Maintenant nous savons ce qu'est le mot de passe réel également.

Notez que j'utilise le bash shell. Et LD\_PRELOAD est une variable d'environnement, ce qui signifie qu'il appartient à votre shell de définir cette variable. Si vous avez des problèmes pour la définir, vous devriez vous reporter au man de votre shell, pour voir comment configurer les variables d'environnement.

## Contrôleur

Usurper les fonctions est amusant. Mais devenir l'emballage d'une fonction est plus puissant. En tant qu'emballage de fonction, nous accepterons les arguments qui sont envoyés à la fonction originale, les passerons à la fonction originale et examinerons les résultats. Ensuite nous pourrons décider de retourner la valeur originelle, ou fixer la valeur de retour. L'appel à la fonction initiale se fera en utilisant un pointeur sur l'adresse initiale de la fonction. Nous allons ensuite utiliser l'API de l'éditeur de liens, c'est la famille de fonction dl\*().

Voici notre nouveau challenge, qui est un peu plus complexe:

```
/*
 * crypt-mix.c
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char **argv) {

char buf[256], alpha[34], beta[34];
int j, plen, fd;

if (argc < 2) {
printf("usage: %s <keyfile>\n", argv[0]);
return 1;
}

if (strlen(argv[1]) > 256) {
fprintf(stderr, "longueur de keyfile > 256, va à la pêche!\n");
return 0;
}

fd = open(argv[1], O_RDONLY);

if (fd < 0) {
perror(argv[1]);
return 0;
}

memset(buf, 0x0, sizeof(buf));

plen = read(fd, buf, strlen(argv[1]));

if (plen != strlen(argv[1])) {

if (plen < 0) {
perror(argv[1]);
}
```

```

printf("Désolé!\n");
return 0;
}

strncpy(alpha, (char *)crypt(argv[1], "$1$"), sizeof(alpha));
strncpy(beta, (char *) crypt(buf, "$1$"), sizeof(beta));

for (j = 0; j < strlen(alpha); j++) {

if (alpha[j] != beta[j]) {

printf("Désolé!\n");
return 0;
}
}

printf("Gagné!\n");

return 1;
}

```

Le principe de ce challenge est relativement simple. Il effectue un hash MD5 sur le nom de fichier donné, puis utilise le même nombre d'octets que la longueur du nom de fichier, et effectue la même fonction MD5 sur le payload.

Puis il compare les deux, sans utiliser explicitement la fonction strcmp(). Cela nous force à trouver un point faible.

Le point faible de ce challenge est la fonction crypt() qui peut être dupée.

Le plan est d'utiliser le 1<sup>er</sup> hash retourné par le 1<sup>er</sup> appel crypt(), puis de fixer la 2<sup>ème</sup> valeur retournée par crypt(), ainsi elle correspondra au 1<sup>er</sup> hash, et passer la partie comparer en douceur.

Voici notre nouvelle bibliothèque partagée:

```

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

/*
 * crypt-mixup.c, Stocke le résultat de crypt() dans un buffer pour le renvoyer plus tard.
 */

// Pointeur sur l'appel crypt() originel
static char *(*_crypt)(const char *key, const char *salt) = NULL;

// Pointeur sur le résultat précédent de crypt()
static char *crypt_res = NULL;

/*
 * crypt, Crooked crypt function
 */

char *crypt(const char *key, const char *salt) {

```

```

// Initialisation de _crypt(), si besoin.
if (_crypt == NULL) {
    _crypt = (char * (*)(const char *key, const char *salt))
        dlsym(RTLD_NEXT, "crypt");
    crypt_res = NULL;
}

// Pas de résultat précédent, continue comme la crypt() normale
if (crypt_res == NULL) {
    crypt_res = _crypt(key, salt);
    return crypt_res;
}

// On a déjà le résultat dans un buffer!
_crypt = NULL;
return crypt_res;
}

```

Nous allons débuter en testant simplement le challenge, en faisant:

```

root@magicbox:/tmp# gcc -o crypt-mix crypt-mix.c -lcrypt
root@magicbox:/tmp# echo "foobar" > mykey
root@magicbox:/tmp# ./crypt-mix mykey
Désolé!

```

Maintenant testons le à nouveau avec notre librairie partagée, en faisant:

```

root@magicbox:/tmp# gcc -fPIC -c -o crypt-mixup.o crypt-mixup.c
root@magicbox:/tmp# gcc -shared -o crypt-mixup.so crypt-mixup.o -ldl
root@magicbox:/tmp# LD_PRELOAD="./crypt-mixup.so" ./crypt-mix mykey
Gagné!

```

Une fois de plus, en utilisant LD\_PRELOAD, nous sommes passés outre le mécanisme.

## Cerberus

Après avoir fait quelques astuces de haut niveau, il est temps de nous salir les mains. Et cela signifie faire intervenir l'Assembleur. Le prochain challenge peut paraître un peu bizarre:

```
/*
 * cerberus.c, Déclaration impossible
 */

#include <stdio.h>

int main(int argc, char **argv) {
int a = 13, b = 17;

if (a != b) {
printf("Désolé!\n");
return 0;
}

printf("Gagné \n");
exit(1);
}
```

Comme vous pouvez le voir dans ce challenge, l'entrée ne compte pas. La déclaration sera toujours incorrecte. Indépendamment de n'importe quel paramètre passé à main(). Mais il existe une méthode! Pour comprendre cette astuce, commençons par désassembler la fonction principale (main):

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>: push  %ebp
0x080483c5 <main+1>: mov   %esp,%ebp
0x080483c7 <main+3>: sub   $0x18,%esp
0x080483ca <main+6>: and   $0xffffffff0,%esp
0x080483cd <main+9>: mov   $0x0,%eax
0x080483d2 <main+14>: sub   %eax,%esp
0x080483d4 <main+16>: movl  $0xd,0xfffffff(%ebp)
0x080483db <main+23>: movl  $0x11,0xfffffff8(%ebp)
0x080483e2 <main+30>: mov   0xfffffff(%ebp),%eax
0x080483e5 <main+33>: cmp   0xfffffff8(%ebp),%eax
0x080483e8 <main+36>: je    0x8048403 <main+63>
0x080483ea <main+38>: sub   $0xc,%esp
0x080483ed <main+41>: push  $0x8048560
0x080483f2 <main+46>: call  0x80482d4 <_init+56>
0x080483f7 <main+51>: add   $0x10,%esp
0x080483fa <main+54>: movl  $0x0,0xfffffff4(%ebp)
0x08048401 <main+61>: jmp   0x804841d <main+89>
0x08048403 <main+63>: sub   $0xc,%esp
0x08048406 <main+66>: push  $0x8048580
0x0804840b <main+71>: call  0x80482d4 <_init+56>
0x08048410 <main+76>: add   $0x10,%esp
0x08048413 <main+79>: sub   $0xc,%esp
0x08048416 <main+82>: push  $0x0
```

```

0x08048418 <main+84>: call 0x80482e4 <_init+72>
0x0804841d <main+89>: mov 0xfffffff4(%ebp),%eax
0x08048420 <main+92>: leave
0x08048421 <main+93>: ret
0x08048422 <main+94>: nop
0x08048423 <main+95>: nop
0x08048424 <main+96>: nop
0x08048425 <main+97>: nop
0x08048426 <main+98>: nop
0x08048427 <main+99>: nop
0x08048428 <main+100>: nop
0x08048429 <main+101>: nop
0x0804842a <main+102>: nop
0x0804842b <main+103>: nop
0x0804842c <main+104>: nop
0x0804842d <main+105>: nop
0x0804842e <main+106>: nop
0x0804842f <main+107>: nop
End of assembler dump.
(gdb)

```

La déclaration IF intervient en <main+36>, le 'je' n'est pas évalué. Donc on ne saute pas vers <main+63> mais continuons vers <main+38>. Ici la chaîne est pauser sur la pile et cela appelle printf() – Une seconde! Est-ce que tu penses comme moi? \*RET HIJACK\* ;-)

```

#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
#include <stdarg.h>

/*
 * megatron.c, Rendons l'impossible possible!
 */

// Pointeur sur l'appel printf() originel
static int (*_printf)(const char *format, ...) = NULL;

/*
 * printf, Une fonction moche!
 */

int printf(const char *format, ...) {
    if (_printf == NULL) {
        _printf = (int (*)(const char *format, ...))
            dlsym(RTLD_NEXT, "printf");

        // Usurpe l'adresse de RETour et la change en <main+66>
        __asm__ __volatile__ (
            "movl 0x4(%ebp), %eax \n"
            "addl $15, %eax \n"
            "movl %eax, 0x4(%ebp)"
        );
    }
}

```

```

return 1;
}

// Rewind ESP and JMP into _PRINTF()
__asm__ __volatile__ (
"addl $12, %%esp \n"
"jmp *%0 \n"
: /* no output registers */
: "g" (_printf)
: "%esp"
);

/* NEVER REACH */
return -1;
}

```

Comme toujours avant l'attaque, nous testons le challenge, comme ceci:

```

root@magicbox:/tmp# ./cerberus
Désolé!

```

It is a pretty straight forward challenge. Maintenant attaquons ce challenge avec notre bibliothèque partagée:

```

root@magicbox:/tmp# gcc -fPIC -o megatron.o megatron.c -c
root@magicbox:/tmp# gcc -shared -o megatron.so megatron.o -ldl
root@magicbox:/tmp# LD_PRELOAD="./megatron.so" ./cerberus
Gagné!

```

Notre attaque a réussi. La manipulation de l'adresse de retour de la fonction printf() a rendu possible l'impossible. Je ne vais pas rentrer dans les détails de cette attaque, car cela pourrait être l'objet d'un document à elle seule. Je dois dire que j'ai triché un peu, car ce code a provoqué une corruption de la pile, et j'ai utilisé la fonction exit() pour faire le travail ingrat pour moi. Notice that the printf() family functions are unusual in the way they meant to accept unlimited number of parameters. Le code ci-dessus se veut être une démonstration (proof of concept) de comment il est possible de manipuler le flux du programme dynamiquement.

Pour conclure ce document, je dois dire que LD\_PRELOAD est une fonctionnalité puissante, et peut être utilisée pour plusieurs finalités et le Reverse Engineering n'en est qu'une.

Contact:

Izik <izik@tty64.org> [or] <http://www.tty64.org>  
 Jerome Athias <jerome.athias@free.fr> [ou] <http://www.athias.fr>