LD_PRELOAD基础用法

By ivan.L

发表于 2018-04-08

本章介绍一下 `LD_PRELOAD` 相关知识，在此做一个记录以便后续查阅。当前操作系统环境为：

```
# cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
```

1. LD_PRELOAD介绍

Linux操作系统的动态链接库在加载过程中，动态链接器会先读取 `LD_PRELOAD` 环境变量和默认配置文件 `/etc/ld.so.preload` ，并将读取到的动态链接库文件进行预加载。即使程序不依赖这些动态链接库， `LD_PRELOAD` 环境变量和 `/etc/ld.so.preload` 配置文件中指定的动态链接库依然会被加载，因为它们的优先级比LD_LIBRARY_PATH环境变量所定义的链接库查找路径的文件优先级要高，所以能够提前于用户调用的动态库载入。

简单来说LD_PRELOAD的加载是最优先级的我们可以用他来做一些有趣的操作(骚操作).

> 一般情况下， `ld-linux.so` 加载动态链接库的顺序为：
>
> LD_PRELOAD > LD_LIBRARY_PATH > /etc/ld.so.cache > /lib > /usr/lib

2. 测试阶段

1) 编写初始rund程序

编写如下文件 `rund.c` ：

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>


int main(int argc, char *argv[])
{
        int i;

        srand(time(NULL));
        for(i=0;i<10;i++){
                printf("%d\n",rand()%100);
        }
        return 0;
```

```
}
```

编译运行：

```
# gcc -o rund rund.c
# ./rund
18
48
5
32
51
80
89
23
20
26
```

2) **创建libmyrand.so动态链接库**

然后写另一个C程序 `myrand.c`：

```
#include<stdio.h>

int rand()
{
        return 55;
}
```

执行如下命令将 `myrand.c` 程序编译为 `libmyrand.so` 文件：

```
# gcc -o libmyrand.so -shared -fPIC myrand.c
# ls
libmyrand.so  myrand.c  rund  rund.c
```

说明：

- `-shared` 是生成共享库格式
- `-fPIC` 选项作用于编译阶段，告诉编译器产生与位置无关代码（Position-Independent Code）；这样一来，产生的代码中就没有绝对地址了，全部使用相对地址，所以代码可以被加载器加载到内存的任意位置，都可以正确的执行。这正是共享库所要求的，共享库被加载时，在内存的位置不是固定的。

3) **使用LD_PRELOAD替换glibc中的rand**

执行如下命令：

```
# LD_PRELOAD=$PWD/libmyrand.so ./rund
55
55
55
55
55
55
55
55
55
55
```

可以看到调用的rand()方法是我们写的rand()方法 这就是所说的 `LD_PRELOAD` 是最优先级 他并没有去调用原本的rand函数。

此外，我们还可以通过 `export LD_PRELOAD=$PWD/libmyrand.so` 写入环境变量，然后执行ldd命令来了解 `rund` 可执行程序所依赖的动态链接库：

```
# export LD_PRELOAD=$PWD/libmyrand.so
# ldd ./rund
        linux-vdso.so.1 =>  (0x00007ffc0fe1e000)
        /root/workdir/libmyrand.so (0x00007f719568b000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f71952bd000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f719588d000)
```

从上面可以看到现在他链接的so文件是我们所写的，那么真正的rand()方法呢？现在我们执行 `export -n LD_PRELOAD=$PWD/libmyrand.so` 来删除环境变量：

```
# export -n LD_PRELOAD=$PWD/libmyrand.so
# echo $LD_PRELOAD
/root/workdir/libmyrand.so
# ldd ./rund
        linux-vdso.so.1 =>  (0x00007ffe531ab000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f2a8443e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f2a8480c000)
```

上面我们看到 `/lib64/libc.so.6` 就是真正的rand方法所在处。

另外一点需要注意的是，当我们执行 `export -n` 后，通过 `echo $LD_PRELOAD` 发现其值并没有发生改变，这是正常的。我们举一个例子：

```
# a=3
# echo $a
```

```
3
```

3. 骚操作

linux中 `woami` 是会调用底层的puts方法。

首先是puts方法，编写如下程序 `who.c` :

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


int puts(const char *message) {
        int (*new_puts)(const char *message);
        int result;
        new_puts = dlsym(RTLD_NEXT, "puts");
        result = new_puts(message);
        return result;
}
```

我们可以在这里添加我们想要执行的代码:

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


int puts(const char *message) {
        int (*new_puts)(const char *message);
        int result;
        new_puts = dlsym(RTLD_NEXT, "puts");
        printf("this is id:%d\n",getuid());     //获取他的uid并输出
        result = new_puts(message);
        return result;
}
```

之后就编译为so文件:

```
# gcc who.c -o who.so -fPIC -shared -ldl -D_GNU_SOURCE
# ls
who.c  who.so
```

- -ldl 显示方式加载动态库，可能会调用dlopen、dlsym、dlclose、dlerror
- -D_GNU_SOURCE 以GNU规范标准编译，如果不加上这个参数会报RTLD_NEXT未定义的错误

然后在把环境变量也加上:

```
# ldd whoami
# whoami
this is id:0
root
```

这个骚操作能用在什么地方就不用了多说了吧.

4. 扩展

系统函数那么多该怎么办?

```
__attribute__((constructor))
    constructor参数让系统执行main()函数之前调用函数(被__attribute__((constructor))修饰的函数)


__attribute__((destructor))
    destructor参数让系统在main()函数退出或者调用了exit()之后,(被__attribute__((destructor))修
```

先测试一下劫持代码 `hijack.c` :

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

__attribute__((constructor)) void jxk() {
        system("ls");
}
```

然后编译和设置环境变量:

```
# gcc -o libhijack.so -shared -fPIC hijack.c
# ls
hijack.c  leveldb  libhijack.so  libmyrand.so  myrand.c  rund  rund.c  who.c  w

# export LD_PRELOAD=$PWD/libhijack.so
```

然后我们写一个简单的程序 `helloworld.c` :

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

```
int main(int argc, char *argv[])
{
        printf("hello, world\n");

        return 0x0;
}
```

编译，然后使用libhijack.so对其进行劫持：

```
# gcc -o helloworld helloworld.c
# export LD_PRELOAD=$PWD/libhijack.so
# ./helloworld
```

当执行后，在我们测试的操作系统环境中(centos 7.5)，我们发现陷入了死循环。这是因为一直在调用所劫持的函数，所以我们修改一下(hijack.c)：

```
#include <stdlib.h>

__attribute__((constructor)) void jxk() {
    if(getenv("LD_PRELOAD") == NULL) return;      //getenv是获取环境变量 当他为空时我们已经不

        unsetenv("LD_PRELOAD");                       //unsetenv删除环境变量的函数 调用一次
    system("ls");
}
```

然后编译和设置环境变量:

```
# gcc -o libhijack.so -shared -fPIC hijack.c
# export LD_PRELOAD=$PWD/libhijack.so
```

然后再执行我们上面的 `helloworld` 程序：

```
# export LD_PRELOAD=$PWD/libhijack.so
# ./helloworld
helloworld  helloworld.c  hijack.c  leveldb  libhijack.so  libmyrand.so  myrand
hello, world
```

上面看到，就只执行了一次 `ls` 命令了。

[参看]:

1. LD_PRELOAD作用
2. LD_PRELOAD基础用法

3. LD_PRELOAD用法

4. What Is the LD_PRELOAD Trick?

5. ld.so(8) — Linux manual page

linux LinuxOps