Author: wzt
EMail: wzt@xsec.org
Site: http://www.xsec.org & http://hi.baidu.com/wzt85
Date: 2008-9-23

一. 概述

目前通用的隐藏文件方法还是 hooksys_getdents64 系统调用， 大致流程就是先调用原始的 sys_getdents64 系统调用，然后在在 buf 中做过滤。修改 sys_call_table 是比较原始的 rk 技术了，
碰到好点的管理员， 基本上 gdb 一下 vmlinux 就能检测出来。 如何想做到更加隐蔽的话，就要
寻找新的技术。 inline hook 也是目前比较流行的做法，不容易检测。本文通过讲解一种利用
inline hook 内核中某函数， 来达到隐藏文件的方法。

二. 剖析 sys_getdnts64 系统调用
想隐藏文件， 还是要从 sys_dents64 系统调用下手。 去看下它在内核中是如何实现的。
代码在 linux-2.6.26/fs/readdir.c 中：

```
asmlinkage long sys_getdents64(unsigned int fd, struct linux_dirent64 __user * dirent, unsigned int count)
{
        struct file * file;
        struct linux_dirent64 __user * lastdirent;
        struct getdents_callback64 buf;
        int error;

        error = -EFAULT;
        if (!access_ok(VERIFY_WRITE, dirent, count))
                goto out;

        error = -EBADF;
        file = fget(fd);
        if (!file)
                goto out;

        buf.current_dir = dirent;
        buf.previous = NULL;
        buf.count = count;
        buf.error = 0;

        error = vfs_readdir(file, filldir64, &buf);
        if (error < 0)
```

```
                goto out_putf;
        error = buf.error;
        lastdirent = buf.previous;
        if (lastdirent) {
                typeof(lastdirent->d_off) d_off = file->f_pos;
                error = -EFAULT;
                if (__put_user(d_off, &lastdirent->d_off))
                        goto out_putf;
                error = count - buf.count;
        }

out_putf:
        fput(file);
out:
        return error;
}
```

首先调用 access_ok 来验证是下用户空间的 dirent 地址是否越界，是否可写。 接着根据 fd，利用 fget 找到对应的 file 结构。 接着出现了一个填充 buf 数据结构的操作，先不管它是干什么的，
接着往下看。
vfs_readdir(file, filldir64, &buf);
函数最终还是调用 vfs 层的 vfs_readdir 来获取文件列表的。 到这，我们可以是否通过 hook vfs_readdir 来达到隐藏文件的效果呢。 继续跟踪 vfs_readdir 看看这个想法是否可行。

源代码在同一文件中：

```
int vfs_readdir(struct file *file, filldir_t filler, void *buf)
{
        struct inode *inode = file->f_path.dentry->d_inode;
        int res = -ENOTDIR;
        if (!file->f_op || !file->f_op->readdir)
                goto out;

        res = security_file_permission(file, MAY_READ);
        if (res)
                goto out;

        res = mutex_lock_killable(&inode->i_mutex);
        if (res)
                goto out;

        res = -ENOENT;
        if (!IS_DEADDIR(inode)) {
```

```
                res = file->f_op->readdir(file, buf, filler);
                file_accessed(file);
        }
        mutex_unlock(&inode->i_mutex);
out:
        return res;
}

EXPORT_SYMBOL(vfs_readdir);
```

它有 3 个参数，第一个是通过 fget 得到的 file 结构指针， 第 2 个通过结合上下文可得知，这是一个
回调函数用来填充第 3 个参数开始的用户空间的指针。 接着看看它具体是怎么实现的。
通过 security_file_permission()验证后， 在用 mutex_lock_killable()对 inode 结构加了锁。
然后调用 ile->f_op->readdir(file, buf, filler);通过进一步的底层函数来对 buf 进行填充。
这个 buf 就是用户空间 strcut dirent64 结构的开始地址。

所以到这里我们可以断定通过 hook vfs_readdir 函数对 buf 做过滤还是可以完成隐藏文件的功能。
而且 vfs_readdir 的地址是导出的， 这样就不用复杂的方法找它的地址了。

但是还有没有更进一步的方法呢？ 前面不是提到过有个 filldir64 函数吗， 它用来填充 buf 结构的。
也许通过 hook 它来做更隐蔽的隐藏文件方法。 继续跟踪 filldir64，看看它是怎么实现的。

```
static int filldir64(void * __buf, const char * name, int namlen, loff_t offset,
                     u64 ino, unsigned int d_type)
{
        struct linux_dirent64 __user *dirent;
        struct getdents_callback64 * buf = (struct getdents_callback64 *) __buf;
        int reclen = ALIGN(NAME_OFFSET(dirent) + namlen + 1, sizeof(u64));

        buf->error = -EINVAL;
        if (reclen > buf->count)
                return -EINVAL;
        dirent = buf->previous;
        if (dirent) {
                if (__put_user(offset, &dirent->d_off))
                        goto efault;
        }
        dirent = buf->current_dir;
        if (__put_user(ino, &dirent->d_ino))
                goto efault;
        if (__put_user(0, &dirent->d_off))
```

```
                goto efault;
        if (__put_user(reclen, &dirent->d_reclen))
                goto efault;
        if (__put_user(d_type, &dirent->d_type))
                goto efault;
        if (copy_to_user(dirent->d_name, name, namlen))
                goto efault;
        if (__put_user(0, dirent->d_name + namlen))
                goto efault;
        buf->previous = dirent;
        dirent = (void __user *)dirent + reclen;
        buf->current_dir = dirent;
        buf->count -= reclen;
        return 0;
efault:
        buf->error = -EFAULT;
        return -EFAULT;
}
```

先把参数 buf 转换成 struct getdents_callback64 的结构指针。

```
struct getdents_callback64 {
        struct linux_dirent64 __user * current_dir;
        struct linux_dirent64 __user * previous;
        int count;
        int error;
};
```

current_dir 始终指向当前的 struct dirent64 结构，filldir64 每次只填充一个 dirent64 结构。它是被 file->f_op->readdir 循环调用的。 通过代码可以看出是把 dirent64 结构的相关项拷贝到
用户空间的 dirent64 结构中， 然后更新相应的指针。

所以通过分析 filldir64 代码， 可以判定通过判断参数 name，看它是否是我们想隐藏的文件，是的话，return 0 就好了。

三. 扩展

通过分析 sys_getdents64 代码的实现，我们可以了解到通过 hook 内核函数的方法，来完成 rootkit 的功能是很简单和方便的。 关键你能了解它的实现逻辑。 对 linux 平台来说，阅读内核
源代码是开发 rootkit 的根本。 如何 hook？ 最简单的就是修改函数的前几个字节，jmp 到我们的
新函数中去， 在新函数完成类似函数的功能。 根本不必在跳回原函数了， 有了内核源代码在手，
原函数怎么实现，我们就怎么 copy 过去给它在实现一次。 所在 linux 实现 rk 也有很方便的

一点，
就是它的内核源代码是公开的， 好好阅读源代码吧， 你会有更多的收获。