LKLAB05

1 劫持syscall_open

首先按照LKMPG-8.1的方式，编写syscall.c，试图安装一个内核模块，修改syscall_table，将地址改为指向自己编写的函数。

问题1：syscall_table在最新的内核版本中不再导出

解决方案：

- 根据文档的提示，从LKMPG的源码处https://github.com/hamjam/lkmpg-examples下载patch，修改内核源码重新编译并替换内核。但是由于版本变化，patch已经无法匹配现在的内核源码，于是查看patch代码，对应地手动修改内核源码。

```
// patch
--- kernel/kallsyms.c.orig  2003-12-30 07:07:17.000000000 +0000
+++ kernel/kallsyms.c   2003-12-30 07:43:43.000000000 +0000
@@ -184,7 +184,7 @@
         iter->pos = pos;
         return get_ksymbol_mod(iter);
     }
-
+
     /* If we're past the desired position, reset to start. */
     if (pos < iter->pos)
         reset_iter(iter);
@@ -291,3 +291,11 @@

 EXPORT_SYMBOL(kallsyms_lookup);
 EXPORT_SYMBOL(__print_symbol);
+/* START OF DIRTY HACK:
+ * Purpose: enable interception of syscalls as shown in the
+ * Linux Kernel Module Programming Guide. */
+extern void *sys_call_table;
+EXPORT_SYMBOL(sys_call_table);
+ /* see http://marc.free.net.ph/message/20030505.081945.fa640369.html
+  * for discussion why this is a BAD THING(tm) and no longer supported by 2.6.0
+  * END OF DIRTY HACK: USE AT YOUR OWN RISK */
```

- 修改kernel/kallsyms.c，在相应位置加上：

```
extern void *sys_call_table;
EXPORT_SYMBOL_GPL(sys_call_table);
```

- 由于我用的是EXPORT_SYMBOL_GPL，因此要在syscall.c加上：

```
MODULE_LICENSE("GPL");
```

重新编译替换内核之后，编译LKMPG的syscall.c，insmod。

问题2：加载模块后进程直接被Kill

解决方案：

- 原因是sys_call_table的权限是只读，因此在修改之前需要修改CR0寄存器获得写权限
- 在syscall.c的开头加上：

```
#define CR0_WP 0x00010000

unsigned long cr0;
```

- 在init_module函数和clean_module函数中修改sys_call_table的前后加上三行代码：

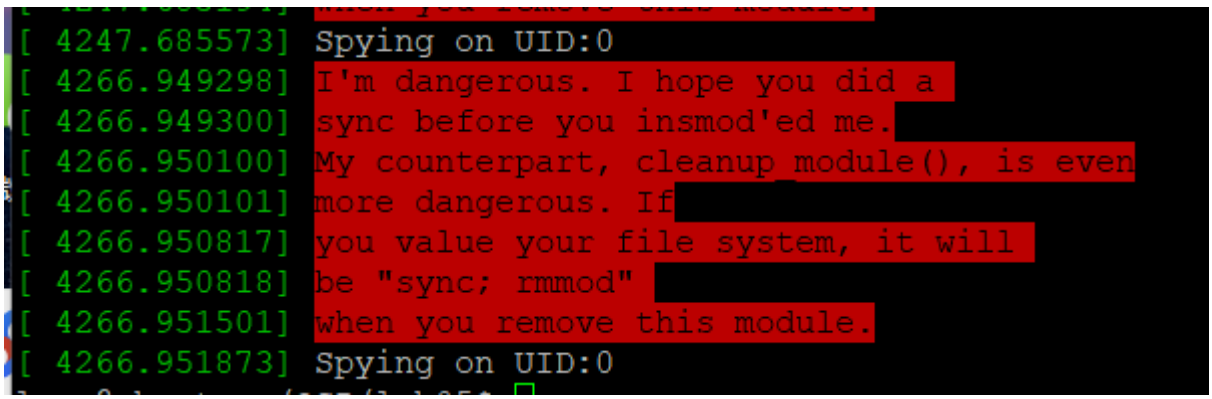
```
cr0 = read_cr0();
write_cr0(cr0 & ~CR0_WP);

sys_call_table[__NR_open] = original_call;

write_cr0(cr0);
```

结果

```
// 编写syscall.c的内核模块makefile
make
sudo insmod syscall.ko
dmsg
```



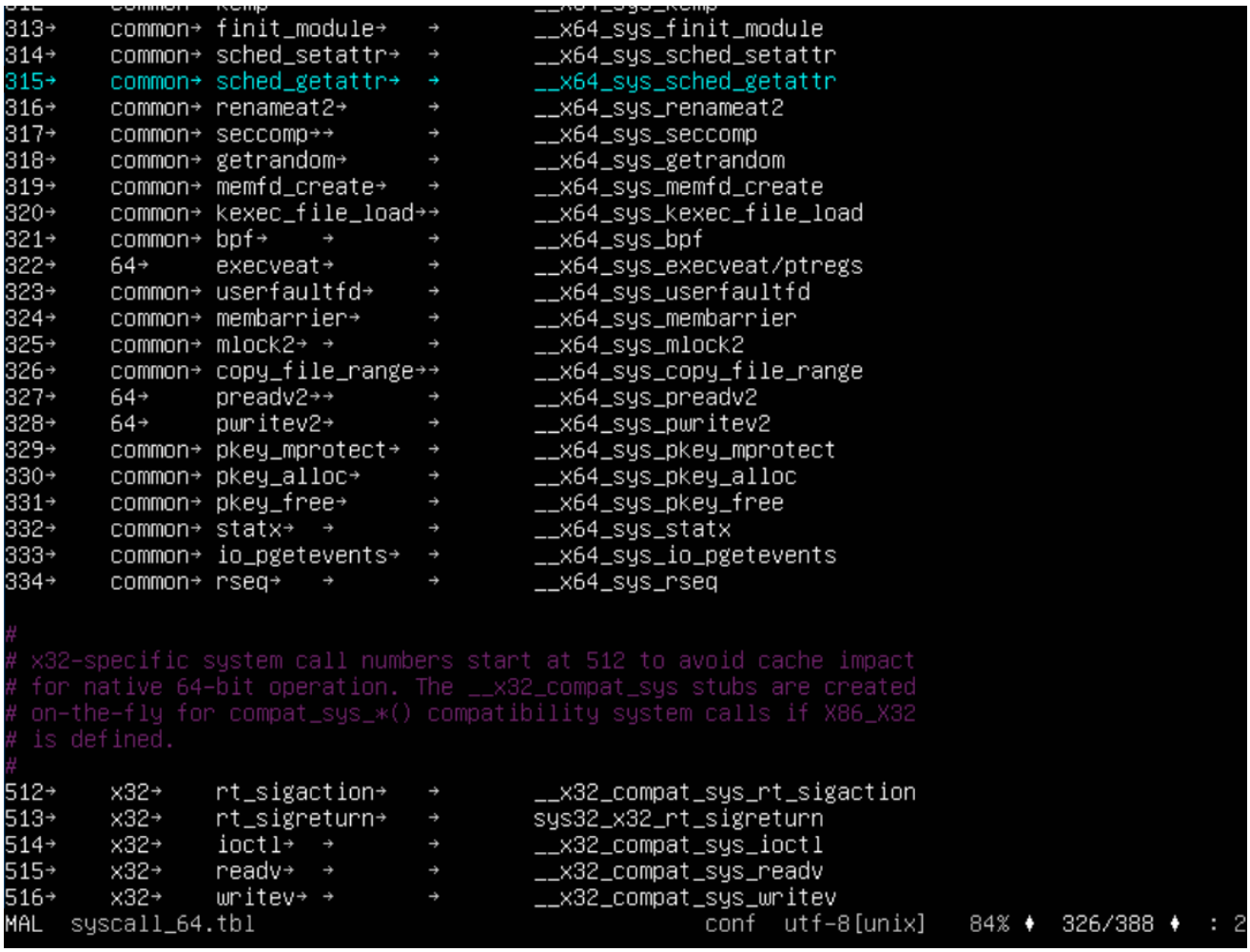
syscall.c源码见文档末尾

2 系统调用

查看源码中的系统调用表，64位的系统调用有335个

```
vim arch/x86/entry/syscalls/syscall_64.tbl
```

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0→      common→ read→   →       →       __x64_sys_read
1→      common→ write→  →       →       __x64_sys_write
2→      common→ open→   →       →       __x64_sys_open
3→      common→ close→  →       →       __x64_sys_close
4→      common→ stat→   →       →       __x64_sys_newstat
5→      common→ fstat→  →       →       __x64_sys_newfstat
6→      common→ lstat→  →       →       __x64_sys_newlstat
7→      common→ poll→   →       →       __x64_sys_poll
8→      common→ lseek→  →       →       __x64_sys_lseek
9→      common→ mmap→   →       →       __x64_sys_mmap
10→     common→ mprotect→       →       __x64_sys_mprotect
11→     common→ munmap→ →       →       __x64_sys_munmap
12→     common→ brk→    →       →       __x64_sys_brk
13→     64→     rt_sigaction→   →       __x64_sys_rt_sigaction
14→     common→ rt_sigprocmask→ →       __x64_sys_rt_sigprocmask
15→     64→     rt_sigreturn→   →       __x64_sys_rt_sigreturn/ptregs
16→     64→     ioctl→  →       →       __x64_sys_ioctl
17→     common→ pread64→→       →       __x64_sys_pread64
18→     common→ pwrite64→       →       __x64_sys_pwrite64
19→     64→     readv→  →       →       __x64_sys_readv
20→     64→     writev→ →       →       __x64_sys_writev
21→     common→ access→ →       →       __x64_sys_access
22→     common→ pipe→   →       →       __x64_sys_pipe
23→     common→ select→ →       →       __x64_sys_select
24→     common→ sched_yield→    →       __x64_sys_sched_yield
MAL  syscall_64.tbl                                  conf  utf-8[unix]    6% ⧫   27/388 ⧫  :  1
```



3 头文件循环依赖

执行make headerdep

```
1   In file included from linux/kernel.h,
  1                  from asm-generic/bug.h:18
  2                  from linux/bug.h:32
  3                  from linux/jump_label.h:190
  4                  from linux/dynamic_debug.h:6
  5                  from linux/printk.h:336
  6                  from linux/kernel.h:13 <-- here
  7 ./include/soc/fsl/dpaa2-fd.h:10: warning: recursive header inclusion
```

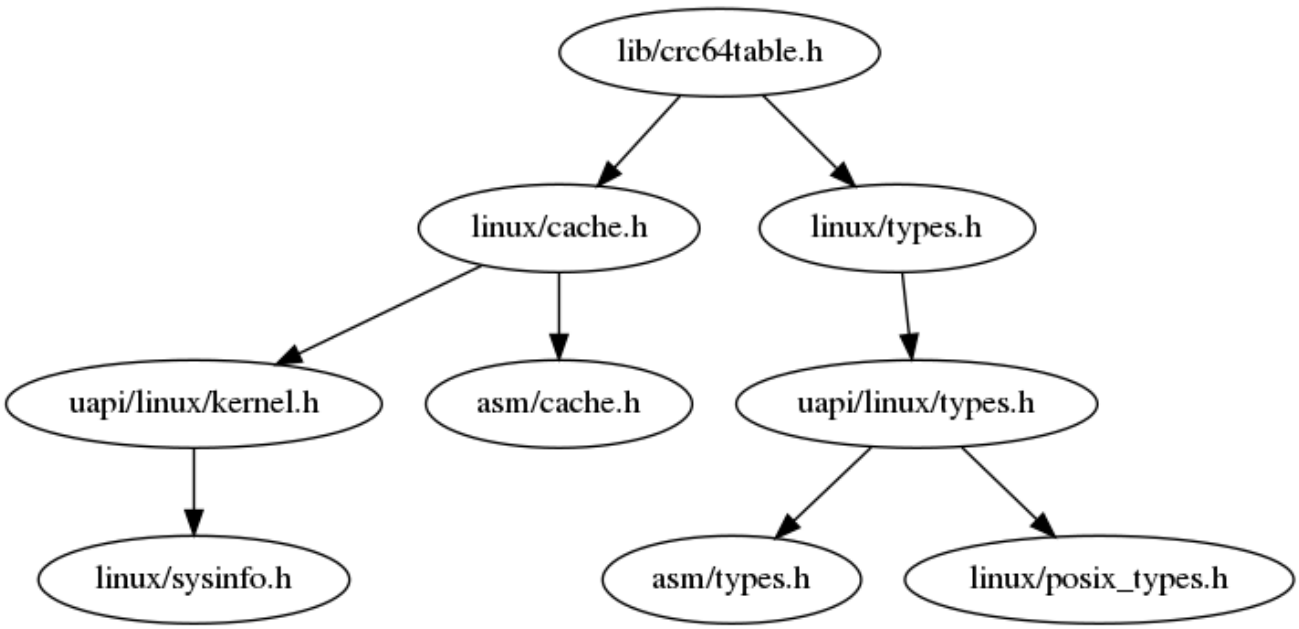
```
// 循环依赖关系
kernel.h -> asm-generic/bug.h -> linux/bug.h -> linux/jump_label.h ->
linux/dynamic_debug.h -> linux/printk.h -> linux/kernel.h
```

将printk.h中的代码复制到kernel.h中去，并删除对dynamic_debug.h的include，再次执行make headerdep时该warning消失

4 头文件调用关系图

使用headerdep.pl绘制lib/crc46table.h头文件调用关系图

```
scripts/headerdep.pl -- graph lib/crc64table.h | dot -Tpng -o graph.png
```



Appendix

```
/*
 *  syscall.c -- source code
 *
 *  System call "stealing" sample.
 */

/*
 * Copyright (C) 2001 by Peter Jay Salzman
 */

/*
 * The necessary header files
 */

/*
 * Standard in kernel modules
 */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module, */
#include <linux/moduleparam.h>  /* which will have params */
#include <linux/unistd.h>   /* The list of system calls */

/*
 * For the current (process) structure, we need
 * this to know who the current user is.
 */
#include <linux/init.h>
#include <linux/types.h>
#include <linux/syscalls.h>
#include <linux/sched.h>
```

```c
#include <asm/uaccess.h>

/*
 * The system call table (a table of functions). We
 * just define this as external, and the kernel will
 * fill it up for us when we are insmod'ed
 *
 * sys_call_table is no longer exported in 2.6.x kernels.
 * If you really want to try this DANGEROUS module you will
 * have to apply the supplied patch against your current kernel
 * and recompile it.
 */
// Write Protect Bit (CR0:16)
#define CR0_WP 0x00010000

// To get EXTERN_SYMBOL_GPL symbols -- sys_call_table
MODULE_LICENSE("GPL");

extern void *sys_call_table[];

/*
 * UID we want to spy on - will be filled from the
 * command line
 */
static int uid;
module_param(uid, int, 0644);
unsigned long cr0;
/*
 * A pointer to the original system call. The reason
 * we keep this, rather than call the original function
 * (sys_open), is because somebody else might have
 * replaced the system call before us. Note that this
 * is not 100% safe, because if another module
 * replaced sys_open before us, then when we're inserted
 * we'll call the function in that module - and it
 * might be removed before we are.
 *
 * Another reason for this is that we can't get sys_open.
 * It's a static variable, so it is not exported.
 */
asmlinkage int (*original_call) (const char *, int, int);

/*
 * The function we'll replace sys_open (the function
 * called when you call the open system call) with. To
 * find the exact prototype, with the number and type
 * of arguments, we find the original function first
 * (it's at fs/open.c).
 *
 * In theory, this means that we're tied to the
 * current version of the kernel. In practice, the
 * system calls almost never change (it would wreck havoc
 * and require programs to be recompiled, since the system
```

```
 * calls are the interface between the kernel and the
 * processes).
 */
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
    int i = 0;
    char ch;

    /*
     * Check if this is the user we're spying on
     */
    if (uid == current->loginuid.val) {
        /*
         * Report the file, if relevant
         */
        printk("uid true:Opened file by %d: ", uid);
        do {
            get_user(ch, filename + i);
            i++;
            printk("%c", ch);
        } while (ch != 0);
        printk("\n");
    }
    else {
            printk("uid false:Opened file by %d: ", current->loginuid.val);
    }

    /*
     * Call the original sys_open - otherwise, we lose
     * the ability to open files
     */
    return original_call(filename, flags, mode);
}

/*
 * Initialize the module - replace the system call
 */
int init_module()
{
    /*
     * Warning - too late for it now, but maybe for
     * next time...
     */
    printk(KERN_ALERT "I'm dangerous. I hope you did a ");
    printk(KERN_ALERT "sync before you insmod'ed me.\n");
    printk(KERN_ALERT "My counterpart, cleanup_module(), is even");
    printk(KERN_ALERT "more dangerous. If\n");
    printk(KERN_ALERT "you value your file system, it will ");
    printk(KERN_ALERT "be \"sync; rmmod\" \n");
    printk(KERN_ALERT "when you remove this module.\n");

    /*
     * Keep a pointer to the original function in
```

```
     * original_call, and then replace the system call
     * in the system call table with our_sys_open
     */
    // change readonly bit
    cr0 = read_cr0();
    write_cr0(cr0 & ~CR0_WP);

    original_call = sys_call_table[__NR_open];
    sys_call_table[__NR_open] = our_sys_open;

    write_cr0(cr0);

    /*
     * To get the address of the function for system
     * call foo, go to sys_call_table[__NR_foo].
     */

    printk(KERN_INFO "Spying on UID:%d\n", uid);

    return 0;
}

/*
 * Cleanup - unregister the appropriate file from /proc
 */
void cleanup_module()
{
    /*
     * D
     * Return the system call back to normal
     */
    if (sys_call_table[__NR_open] != our_sys_open) {
        printk(KERN_ALERT "Somebody else also played with the ");
        printk(KERN_ALERT "open system call\n");
        printk(KERN_ALERT "The system may be left in ");
        printk(KERN_ALERT "an unstable state.\n");
    }
    cr0 = read_cr0();
    write_cr0(cr0 & ~CR0_WP);

    sys_call_table[__NR_open] = original_call;

    write_cr0(cr0);
}
```