LIDS Hacking HOWTO

Xie Huagang, `xie@lids.org`, `http:`/www.lids.org/

v 1.0, 29 March 2000

This Document is for LIDS (Linux Intrusion Detection System) Project and mainly about the impletementation of LIDS in the kernel. The target of the LIDS is to enhance the current Linux Kernel, to make it more solidate and more secure. In order to accomplish the target, LIDS uses the current Linux kernel resouce and adds on some security features. In this HOWTO, I will say something about the LIDS internal to those who concern the inplementation of LIDS. If you want to know how to use LIDS, please refer to `LIDS HOWTO`.

Contents

1 Introduction

With increasing popularity of Linux on Internet , more and more security holes are found in the current GNU/Linux system. You may hear from the Internet that –ooh, There are bugs found in Linux, which will cause the system to be easily compromised by hacker.

Since the Linux is an art of open source community, security holes can be found easily and can also be patched quickly. But when the hole is disclose to the public, and the administrator is too lazy to patch the hole. It is very easy to break into the current system and it is worse that the hacker can get the root shell. With the current GNU/linux system, he can do whatever he want. Now, you may ask, what is the problem and what can we do?

1.1 What is wrong with the current GNU/Linux system.

- superuser (root) may abuse the rights

  Being a root, he can do whatever he want. Even the capability existing in the current the system. As a root, he can easily change the capability.

- Many system files can be changed easily.

  There are many important files, such as /bin/login, in the system. if the hacker come in, he can upload a changed login program to replace /bin/login , so he can relogin without any login name of password. But the files do not need to chang frequently, unless you want to upgrade the system.

- Modules is easily used to intercept the kernel.

  Module is a good design for the linux kernel to make the linux kernel more modulized and more felixible. But after the modules inserted into the kernle, it will be part of the kernel and can do what the original kernel can do. Therefore some unfriendly code could be written as a modules and inserted into to kernel, the code can even redirect the system call and act like a virus.

- Process is unprotected.

  Certain processes, such as web server daemon, which are critical to to system is not under strict protection. Therefore, there are vulnerable to the attack of hackers.

1.2 What is the idea behide LIDS.

- To protect important files.

  Since the files can be easily changed by root, why not restrict the file operation ? so, LIDS change the file system call in the kernel to enhance security. Every time someone access a file, he will involve in a system call and then we can check the filename and see whether the file has been protected or not. If it has been protected, we can then deny the request of the visitor.

- To protect important process.

  There is a little bit different from the above idea to protect the process. When a process is running in the system, he will have an entry to the /proc filesystem with a **pid** as diretory name, so if you type "ps -axf" you can display the current running processes.

  You may wonder how to protect certain process. If you want to kill a process, firstly, you type "ps" to get the process's PID, secondly, you type "kill <pid>" to kill it.

  If I makethe process invisible to you, how can you kill the process? Therefore, what LIDS does to protect the process is to hide the process from everybody.

  Another important method is to protect some important process is to make it unkillable by anybody, include root. LIDS can protect the process whose parent is init (pid = 1).

- To seal the kernel.

  Since we want to insert some necessary modules into the kernel for use , we also don't allow anybody include root to insert the modules. How to balance this to pardox things. Well, we can use the concept provided and first impletemetated by John Carol Langford (`jcl@gs176.sp.cs.cmu.edu`). We just allow the system insert the modules into the system while the system boot up, then we seal the kernel, after sealing, the kernel do not allow anyone to insert modules into the kernel. Using the seal concept, we can use it to protect the important files, process — we just change the necessary files or run the necessary process while the system is booting up, and after sealing the kernel, we can not make any change on the files again.

2 Protect File System

The most important job LIDS do is to protect the file system. It is impletementated in the VFS(virtual File System) layer in the kernel.For that reason, we can protect any kind of filesystem, such as EXT2, FAT.

2.1 Protect important files.

In the LIDS, the protected files are devided into catelog shown below,

- Read Only Files/Diretory.

  Read only files means that they do not allow any changer, for example, files in the diretory of `usr`sbin/, `sbin`.

  This kind of files mostly are binary system program or system configuration files, we do not need to change them unless system upgrade.

- Append Only Files/Directory.

  The Append files catalog is only for the files whose size only allow to grow . Most system log files, such as files in /var/log, are append only files.

- Exeception Files/Directory.

  The files is not to protected. In some case, you want to protect the whole directory but also want some specified file to be unprotected, so we can defined the files as exeception and the diretory as read-only.

- Protection mouting/umouting files system .

  When you mount filesystem after system boot up, you can disable anybody, include root, to umount the filesystem. You can also disable anybody to mount a filesystem to overlap the current filesystem.

2.2 How LIDS protect files in kernel.

In this section, we will view some kernel source let you understand how LIDS protect files.

2.2.1 Linux Filesystem data struct and routines.

Firstly, we must understand the virtual filesytem in linux.

Every file in the linux, whatever the filesytem he reside on, has an inode number, the files system provide the following data struct.

```
in the /usr/src/linux/include/linux/fs.h

struct inode {
        struct list_head        i_hash;
        struct list_head        i_list;
        struct list_head        i_dentry;

        unsigned long           i_ino; ----> inode number.
        unsigned int            i_count;
```

```
            kdev_t                  i_dev; ----> device number.
            umode_t                 i_mode;
            nlink_t                 i_nlink;
            uid_t                   i_uid;
            ......
    }
```

Note: <i_ino ,i_dev> used to be a identification of an inode.That means that you can use the pair <i_ino,i_dev> to get a unique inode from the system.

```
    in the /usr/src/linux/include/linux/dcache.h

    struct dentry {
            int d_count;
            unsigned int d_flags;
            struct inode  * d_inode;        /* Where the name belongs to - NULL is negative */
            struct dentry * d_parent;       /* parent directory */
            struct dentry * d_mounts;       /* mount information */
            struct dentry * d_covers;
            struct list_head d_hash;        /* lookup hash list */
            struct list_head d_lru;         /* d_count = 0 LRU list */
            struct list_head d_child;       /* child of parent list */
            struct list_head d_subdirs;     /* our
            ......
    }
```

The dentry is a entry of a file in diretory. Using the dentry, we can easily trave through the file's parent diretory.

For example, if a file's inode is (struct inode *)file_inode, you can use `file_inode->d_entry` to get its diretory entry and then use `file_inode->d_entry->d_parent` to get its parent diretory's entry.

2.2.2 LIDS Data Struct for protection.

After the above analyzing of the linux file system, let's have a look at how LIDS uses the VFS to protect files and diretories.

```
    /* in /usr/src/linux/fs/lids.c */

    struct secure_ino {
            unsigned long int ino;          /* the inode number */
            kdev_t  dev;                    /* the dev number */
            int     type;                   /* the file type */
    };
```

The above struct is used to store the protected file or directorie's inode with the pair <ino,dev>."type" is used to indentify which type the protected inode(file).

LIDS has 4 type,

```
    /* in /usr/src/linux/incluce/linux/fs.h */
```

```
#define LIDS_APPEND     1       /* APPEND ONLY FILE */
#define LIDS_READONLY   2       /* Read Only File */
#define LIDS_DEVICE     3       /* Protect MBR Writing to device */
#define LIDS_IGNORE     4       /* Ignore the protection */
```

With the secure_ino struct, we can easily initial the protected files or diretories into the kernel with the flollowing functions,

```
/* in /usr/src/linux/fs/lids.c */

int lids_add_inode(unsigned long int inode ,kdev_t dev , int type)
{

        if ( last_secure == (LIDS_MAX_INODE-1))
                return 0;

        secure[last_secure].ino = inode;
        secure[last_secure].dev = dev;
        secure[last_secure].type = type;

        secure[++last_secure].ino = 0;

#ifdef VFS_SECURITY_DEBUG
        printk("lids_add_inode : return %d\n",last_secure);
#endif
        return last_secure;
}
```

As you can see from the above code, it is very easy to add a given inode into the secure_ino. Protected inodes are initialized during the system boot. The initial routine is `init_vfs_security()` in /usr/src/linux/fs/lids.c.

And now, let's have a look at how the LIDS check whether an inode is being protected,

```
/* /usr/src/linux/fs/open.c */

int do_truncate(struct dentry *dentry, unsigned long length)
{
        struct inode *inode = dentry->d_inode;
        int error;
        struct iattr newattrs;

        /* Not pretty: "inode->i_size" shouldn't really be "off_t". But it is. */
        if ((off_t) length < 0)
                return -EINVAL;

#ifdef CONFIG_LIDS
        if (lids_load && lids_local_load) {
                error = lids_check_base(dentry,LIDS_READONLY);
                if (error) {
                        lids_security_alert("Try to truncate a protected file (dev %d %d,inode %ld)",
```

```
                                                MAJOR(dentry->d_inode->i_dev),
                                                MINOR(dentry->d_inode->i_dev),
                                                dentry->d_inode->i_ino);
             .....................
```

This is an example that LIDS add checking in kernel. You can see that the function `lids_check_base()` is one of core functions for the lids protection method.

You can see `lids_check_base()` in many place where LIDS want to protect, especially in fs subdiretory of linux kernel.

```
     /* in /usr/src/linux/fs/lids.c */

     int lids_check_base(struct dentry *base, int flag)
     {
             ..................
             inode  = base->d_inode;         /* get the inode number */
             parent = base->d_parent;        /* get the parent diretory */

             .................
     ---->   do {
                     if ( inode == parent->d_inode)
                             break;
                     if ((retval = lids_search_inode(inode))) {
                             if ( retval == LIDS_IGNORE ||
                                     (retval == LIDS_DEVICE && flag != LIDS_DEVICE))
                                     break;
                             if ( flag == LIDS_READONLY ||
                                     ( flag == LIDS_APPEND && retval >flag ) ||
                                     ( flag == LIDS_DEVICE && flag == retval )) {
                                     return -EROFS;
                             }
                             break;
                     }
                     inode = parent->d_inode;
             } while( ((parent = parent->d_parent ) != NULL) );

             return 0;
     }
```

`lids_check_base()` check if the given dentry of a file and it's parent diretories have been protected.

Note: If the its parent diretory is protected, the file is also protected.

For example, if "/etc/" has been protected, the "/etc/passwd" is also protected.

2.2.3 Protected system call in the kernel.

In order to protect the filesystem, LIDS insert a checking in the begin of some critical system call. Therefore, we can protect the system call and restrict user using the filesystem.

Here we list some example,

- open(), open is used for protect open some files protected with the disallowed permission. You can see the lids checking code in open_namei() when open call.
- mknod(), mknod is used to protect the mknod in the giving diretory.
- unlink(). check do_unlink() in kernel source.

3 Protect Devices , Raw I/O access.

Devices in linux are present as files in "/dev/", we can use the method of protecting files above to protect devices. But in some case, user can also use the IO operation to bypass the filesystem to read/write the device, we must consider that case.

3.1 Device, Raw I/O in kernel

Devices in GNU/Linux system are present as files, so we can protect it using the same method as protecting filesystem.

User space raw I/O access is proformed by the system call `sys_operm` and `sys_iopl`. You can have a look at /usr/src/linux/arch/i386/kernel/ioport.c. This is a archetecture dependence and if we port to other hardware platform, we need to take care about that.

3.2 How to protect by LIDS.

Most of time, application do not need to access the device via the device file name in "/dev/". But some paticular application need to access it directly, such as the X Server, which will write to the /dev/mem and even raw I/O. We need some exception when protect the device. LIDS define the exeception when configurate the Kernel.

- CONFIG_LIDS_ALLOW_DEV_MEM, If you selecte it on, you can allow some specified program to access the /dev/mem and /dev/kmem which is very critical for the kernel. If you want to use X Server System in the kernel, select this and provide the whole path and filename when configurate the kernel.
- CONFIG_LIDS_ALLOW_RAW_DISKS, If you select it on, you can allow some specified program to access the raw disk.
- CONFIG_LIDS_ALLOW_IO_PORTS, If you selecte it on, you can allow some specified program to access the I/O port.

The `initialization` is called when the system in `init_vfs_security()` in`fs/lids.c`.

```
#ifdef CONFIG_LIDS_ALLOW_DEV_MEM
        lids_fill_table(allow_dev_mem,&last_dev_mem,LIDS_MAX_ALLOWED,CONFIG_LIDS_DEV_MEM_PROGS);
#endif

#ifdef CONFIG_LIDS_ALLOW_RAW_DISKS
        lids_fill_table(allow_raw_disks,&last_raw_disks,LIDS_MAX_ALLOWED,CONFIG_LIDS_RAW_DISKS_PROGS);
```

```
#endif

#ifdef CONFIG_LIDS_ALLOW_IO_PORTS
        lids_fill_table(allow_io_ports,&last_io_ports,LIDS_MAX_ALLOWED,CONFIG_LIDS_IO_PORTS_PROGS);
#endif
```

And then, when a process(program) want to access the io port or raw disks directly, LIDS will check if it is an exeception defined in arrary (allow_raw_disks , last_io_ports,etc.). The checking is performed by `lids_search_inode(inode)` which is called by `lids_check_base()`.

For example, let's look at the CONFIG_LIDS_ALLOW_DEV_MEM.

```
/* in lids_search_inode() */

#ifdef CONFIG_LIDS_ALLOW_DEV_MEM
        for( i = 0 ; i < last_dev_mem ;i++ ) {
                if ( allow_dev_mem[i].ino == ino && allow_dev_mem[i].dev == dev) {
                        return LIDS_READONLY;
                }
        }
#endif
#ifdef CONFIG_LIDS_ALLOW_RAW_DISKS
```

We can see that the `allow_dev_mem` contains the inodes of the allowed programs which are initialized in the `init_vfs_security()` when booting. Using the same method, we can protect raw device ,I/O access, etc, except some specified programs.

4 Protect Important Process.

Processes are the active entries in the operation system. There are two specified process in the kernel, process id 0 (swapd) and process 1(init). The init process is parent of all process runing after the system booting.

4.1 unkillable process.

As you can see that if someone gains the root privilege, he can easily kill any process by sending special signal to that process . In order to kill a process, he must past the pid of the process in the kernel, and then use the pid to kill it.

The system call used to kill process is `kill()`, which is impletemented`sys_kill()` in kernel.

Let's look at the code with LIDS protection.

```
/* in /usr/src/linux/kernel/signal.c */
asmlinkage int
sys_kill(int pid, int sig)
{
        struct siginfo info;

#ifdef CONFIG_LIDS_INIT_CHILDREN_LOCK                                           pid_t this_pid;
        int i;
```

```
#ifdef CONFIG_LIDS_ALLOW_KILL_INIT_CHILDREN
        if (!(current->flags & PF_KILLINITC))
#endif
        if (lids_load && lids_local_load && LIDS_FISSET(lids_flags,LIDS_FLAGS_LOCK_INIT_CHILDREN)) {
                this_pid = pid>0?pid:-pid;
                for(i=0;i<lids_last_pid;i++) {
                        if( this_pid == lids_protected_pid[i]) {
                                lids_security_alert("Try to kill pid=%d,sig=%d\n",pid,sig);
                                return -EPERM;
                        }
                }
        }
#endif
...
}
```

You can see that there are two tags in kernel,`CONFIG_LIDS_INIT_CHILDREN_LOCK` and `CONFIG_LIDS_ALLOW_KILL_INIT_CHILDREN`.

With the CONFIG_LIDS_INIT_CHILDREN_LOCK on, LIDS can protect the initial runing up process. For example, if you running inetd on the system, you bring it up before sealing the kernel. After that, you can not kill it. But if someone telnet to the server, inetd will bring up a child process to serve the user, this child process can not be protect by LIDS, because the user can be exit and then kill the process at any time.

4.2 hidden process.

Another method to protect process is to hide the process. when a hacker compromise you system, he will login, and then look around if there are some known processes are watching him, and then he will kill them. If you hide the process by this feature, the hacker will not know anything about the process and then you can log anything he done on the system.

4.2.1 How to hide process.

In order to hide the process, you need to provide the full pathname while configurate the kernel.

When the kernel boot up, LIDS will pick up the filename's inode into a structure named proc_to_hide[],

```
/* include/linux/sched.h */

#ifdef CONFIG_LIDS_HIDE_PROC
#define PF_HIDDEN       0x04000000      /* Hidden process */
#endif

/* in fs/lids.c */

#ifdef CONFIG_LIDS_HIDE_PROC
struct allowed_ino proc_to_hide[LIDS_MAX_ALLOWED];
int last_hide=0;
#endif
....
```

```
/* in fs/lids.c , init_vfs_security(),
   fill up the hidden process in proc_to_hide[]
 */
#ifdef CONFIG_LIDS_HIDE_PROC
        lids_fill_table(proc_to_hide,&last_hide,LIDS_MAX_ALLOWED,CONFIG_LIDS_HIDDEN_PROC_PATH);
#endif
```

PF_HIDDEN is used to make the kernel to check if the process can be seen when user issue a "display process information" command, for example the command "ps -a". If a process has been marked as hidden process by LIDS, when it exec, the process will get an attribute of PF_HIDDEN. Then, when the system output processes information to the user program, it will check if the current output process own a flag PF_HIDDEN. If found, it will not output any information about the process.

```
/* in fs/exec.c */
int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
        ...
        if (retval >= 0) {
#ifdef CONFIG_LIDS_HIDE_PROC
                        if (lids_search_proc_to_hide(dentry->d_inode))
                                current->flags |= PF_HIDDEN;
        ...
```

Since every process in linux has a entry in /proc filesystem, we also need to make the proc file entry of the hidden process insivible to users.

```
/* fs/proc/root.c , to make the proc fs invisibe to the hidden process*/
static struct dentry *proc_root_lookup(struct inode * dir, struct dentry * dentry)
{
...
        inode = NULL;

#ifdef CONFIG_LIDS_HIDE_PROC
        if ( pid && p && (! ((p->flags & PF_HIDDEN) && lids_load && lids_local_load)) ) {
#else
        if (pid && p) {
#endif
                unsigned long ino = (pid >> 16) + PROC_PID_INO;
                inode = proc_get_inode(dir->i_sb, ino, &proc_pid);
                if (!inode)
                        return ERR_PTR(-EINVAL);
                inode->i_flags|=S_IMMUTABLE;
        }
...
}
```

Then if the process is marked as PF_HIDDEN, it will not display in the proc filesystem.

5 Sealing the Kernel.

We need do some necessary operation when system booting up, but also we need to provent the operation when system is running.

For example, we need to insert the needed modules into the kernel, but we dont want any module to be inserted when the system is running because it is very dangerous. How to sulute this problem? here comes the seal idea. we can do what we want to do during the system booting, and then we seal the kernel. After that, we can not do the same thing as what we can do before sealing. With the sealing ideas, we can solute the problem with modules, we can insert the needed modules into kernel before sealing and don't sallow any modules inserted and deleleted after sealing.

5.1 Sealing kernel with the LIDS.

In order to seal the kernel, in LIDS , we use the command

```
#lidsadm -I -- -CAP_xxx...
```

It can be put in the script which can be run by init when system booting up. The detials about capability is in LIDS HOWTO written by biodi. What the lidsadm do is to communicate with kernel via the file `/proc/sys/lids/locks`.

When you seal the kernel, the lidsadm call `lids_init()` in lidsadm.c.

```
/* in lidsadm.c */

#define LIDS_LOCKS      "/proc/sys/lids/locks"
......
void lids_init(int optind, int argc, char *argv[])
{
......
        if ((fd=open(LIDS_LOCKS,O_RDWR)) == -1) {
                perror("open");
                exit_error (2, "can't open " LIDS_LOCKS);
        }
        if (read(fd,&locks,sizeof(lids_locks_t))==-1) {
                perror("read");
                exit_error (2, "can't read " LIDS_LOCKS);
        }

        lids_set_caps(optind,argc,argv,&locks);

        locks.magic1=LIDS_MAGIC_1;
        .........

        if (write(fd,&locks,sizeof(lids_locks_t))==-1) {
                perror("write");
                exit_error (2, "can't write " LIDS_LOCKS);
        }
.....
}
```

The system call write the new variant *locks* to the LIDS_LOCKS, the kernel will read it via the function `lids_proc_locks_sysctl()`. The `lids_proc_locks_sysctl` will do some sanity check and read the locks from user space and perform the capability changed and then change the value of the sealing variant – `lids_first_time` to 0.

Let's have a look at `lids_proc_locks_sysctl()`. This function is called by kernel when someone read/write the `/proc/sys/lids/locks`.

```
int lids_proc_locks_sysctl(ctl_table *table, int write, struct file *filp,
                            void *buffer, size_t *lenp, int conv, int op)
{
...........


        /* first: check the terminal and the program which access the sysctl */

#ifndef CONFIG_LIDS_REMOTE_SWITCH
        if (current->tty && (current->tty->driver.type != 2) ) {
                lids_security_alert("Try to %s locks sysctl (unauthorized terminal)",
                                    write ? "write" : "read");
                return -EPERM;
        }
#endif
........
       /* second: check wether it is not a timeout period after two many failed attempts */

.......
  if (write) {
                /* Third : check what is submitted (size, magics, passwd) */
                if (*lenp != sizeof(lids_locks_t)) {
                        lids_security_alert("Try to feed locks sysctl with garbage");
                        return -EINVAL;
                }
                if (copy_from_user(&locks,buffer,sizeof(lids_locks_t)))
                        return -EFAULT;
        .......
                if ((lids_first_time) && (!locks.passwd[0])) {
                        .........
                        number_failed=0;
                        if (lids_process_flags(locks.flags)) {
                                cap_bset=locks.cap_bset;
                                lids_security_alert("Changed: cap_bset=0x%x lids_flags=0x%x",cap_t(cap_bset),lids_flags);
                        }
 Change flag here ..-->  lids_first_time=0;
                        .....
                }
```

The function above is to do the real job when sealing the kernel or change the kernel security level. The variant `lids_first_time` is a flag indicating that the current state is before-sealing or after-sealing. After change the requried capability bit, the flag change to 1 indicating that the current state is after-sealing.

Sealing kernel has two tasks, firstly, change the capability bit with the requried parameter, secondly, change the `lids_first_time` flag to 1. After the sealing, the system will not allow change the capability without using lidsadm and a password.

5.1.1 Protect program before sealing the kernel.

Since the state before sealing is dangerous, we should know that the program running before sealing is protected by LIDS. Why ? because we must gurantee that the on-going program can not be changed after sealing. If the files is not protected, someone may change the program and then after reboot, the program can also do harm to the system. Let's look at the code about the warning about the unprotected program running before sealing.

```
int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
        ..........

#ifdef CONFIG_LIDS_SA_EXEC_UP
        if (lids_first_time && lids_load) {
                if (!lids_check_base(dentry,LIDS_READONLY))
#ifdef CONFIG_LIDS_NO_EXEC_UP
                        lids_security_alert("Try to exec unprotected program %s before sealing LIDS",filename);
                        if (dentry)
                                dput(dentry);
                        return -EPERM;
#else
                        lids_security_alert("Exec'ed unprotected program %s before sealing LIDS",filename);
#endif
                }
        }
#endif
......
}
```

You can see that when LIDS protected system is on (lids_load == 1) and the current system is no sealed(lids_firest_time is 1), kernel will check if the current program is under protected by LIDS by lids_check_base(). If it is not protected, it will raised warning message.

6 LIDS with Capability

Capability is a set of bit to indicate what a process can do. In LIDS, we use capability to limite the capability of all process.

6.1 Capability in kernel.

The capability in kernel is a 32-bit long varient to indicate what the current process can do. The difinition of capability in kernel as following shown.

```
/* in include/linux/capability.h */

typedef struct __user_cap_header_struct {
        __u32 version;
        int pid;
} *cap_user_header_t;

typedef struct __user_cap_data_struct {
        __u32 effective;
        __u32 permitted;
        __u32 inheritable;
} *cap_user_data_t;

#ifdef __KERNEL__

/* #define STRICT_CAP_T_TYPECHE

#ifdef STRICT_CAP_T_TYPECHECKS

typedef struct kernel_cap_struct {
        __u32 cap;
} kernel_cap_t;

#else

typedef __u32 kernel_cap_t;

#endif

kernel_cap_t cap_bset = CAP_FULL_SET;
```

Every bit of the kernel_cap_t present a kind of permission(capability). `cap_bset` is the base of all capability set. It's value can be change by writing to `/proc/sys/kernel/cap-bound`.

Look at same file as shown above, you will find something like that:

```
/* in include/linux/capability.h */

/* In a system with the [_POSIX_CHOWN_RESTRICTED] option defined, this
   overrides the restriction of changing file ownership and group
   ownership. */

#define CAP_CHOWN            0

/* Override all DAC access, including ACL execute access if
   [_POSIX_ACL] is defined. Excluding DAC access covered by
   CAP_LINUX_IMMUTABLE. */

#define CAP_DAC_OVERRIDE     1
```

```
/* Overrides all DAC restrictions regarding read and search on files
   and directories, including ACL restrictions if [_POSIX_ACL] is
   defined. Excluding DAC access covered by CAP_LINUX_IMMUTABLE. */

#define CAP_DAC_READ_SEARCH  2
.........
```

You will get what is the exactly meaning of each capability in this file and the LIDS HOWTO written by Boidi.

Each task(process) has three member defined in struct task_struct: `cap_effective, cap_inheritable, cap_permitted`. We have list a varient `cap_bset` which indicate a base capability. It check the whole system and determin what kinds of capabilty to control the system.

In most of the system call implementated in kernel, they will call the function `capable()`(in kernel/sched.c) in the begin of that function. It then call `cap_raised()` (in include/linux/capability.h) which is shown below,

```
#ifdef CONFIG_LIDS_ALLOW_SWITCH
#define cap_raised(c, flag)  ((cap_t(c) & CAP_TO_MASK(flag)) && ((CAP_TO_MASK(flag) & cap_bset) || (!lids_load) || (!lids_local_
#else
#define cap_raised(c, flag)  (cap_t(c) & CAP_TO_MASK(flag) & cap_bset)
#endif
```

You will see that the `cap_bset` (it is all "1" by default) is very important here. If we set some bits to 0 in it, the capabilties they present will be disable in the whole system. For example, 18th bit represent `CAP_SYS_CHROOT`, if we set it to "0", it means that we can not `chroot` anymore.

If you look at the source code of `sys_chroot`, you will find that it will check the capability by following,

```
if (!capable(CAP_SYS_CHROOT)) {
                goto dput_and_out;
        }
```

The `capable()` will return 0 for the bit 18 is 0 and then the chroot will return an error to the user.

6.2 Capability in LIDS.

LIDS use capability to restrict the whole process's action. The function used by LIDS is `capable()`. With many existing `capable()` in the kernel source, we can disable some capabilities default by the current system and also give some warning when user violate the rules defined by LIDS.

As for the administrator, he can also change the capability by `lidsadm` and a password. After the kernel authenticate the user, the capability variant `cap_bset` will be changed.

The most important thing for an administrator is to understand what exactly each capabilitiy means and know what he want on the system. After that, disable the capabilities when sealing the kernel and change them on the fly with a password.

7 LIDS Security Level in kernel.

At some time, we need to change the protected system, How can we do? LIDS provide two way.

- First, you can reboot the system and enter "security=0" when "LILO:" appear on the screen.
- Secondly, you can also switch the security level online by giving a password with the command "lidsadm -S".

7.1 two levels in the kernel.

LIDS defines two levels in kernel, security and none_security. By default, the security is on. If you want to change it, enter "security=0" after reboot the system.

There is a grobal variant in the kernel name `lids_load`, it indicates whether the lids security system is on or not. It is default to "1" (on) by default. If you input "security=0" when LILO appear, the lids_load will set to "0", and all security protection by LIDS will be bypassed. It is like the system without LIDS protection.

```
/* variant defined in fs/lids.c */
int lids_reload_conf=0;
int lids_load=0;        /* it is raised to 1 when kernel boot */
int lids_local_on=1;
int lids_local_pid=0;

/* in init/main.c */
#ifdef CONFIG_LIDS
/*
 *      lids_setup , read lids info from the kernel.
 */
static void __init lids_setup(char *str, int *ints)
{
        if (ints[0] > 0 && ints[1] >= 0)
    ====>       _lids_load= ints[1];
}
#endif
....
/* init the LIDS when the system bootup up */

static void __init do_basic_setup(void)
{
......
        /* Mount the root filesystem.. */
        mount_root();

#ifdef CONFIG_LIDS
        /* init the ids file system */
--->    lids_load=_lids_load;
        lids_local_on=_lids_load;
        lids_flags=lids_load * (LIDS_FLAGS_LIDS_ON | LIDS_FLAGS_LIDS_LOCAL_ON);
===>    printk("Linux Intrusion Detection System %s \n",lids_load==1?"starts":"stops");
        init_vfs_security();
```

```
#endif
......
}
```

When the system boots up, you can see " Linux Intrusion Detection System 0.9 starts " when lids protection switch on or "Linux Intrusion Detection System 0.9 stops" when the security off. "0.9" is the current LIDS version.

7.2 Change security level online with lidsadm.

At some times, you may also want to change the security level online, you must turn the `CONFIG_LIDS_ALLOW_SWITCH` on and also provide a the "RipeMD-160 encrypted password" field when configurate the kernel before compiles.

The password can be obtained by running the command "lidsamd -P".

7.2.1 Authentication with kernel.

With the provided password, LIDS can use authenticate the user who can switch the kernel security level on and off.

It is also performed by lidsadm with parmeter "-S", for example,

```
# /sbin/lidsadm -S -- -LIDS
SWITCH
Password:xxxxxx
#
```

After input the correct password, you can swith the lids security off.

Let's look at the code internal to see how it does,

```
/* in the fs/lids.c lids_proc_locks_sysctl() */
int lids_proc_locks_sysctl(ctl_table *table, int write, struct file *filp,
                            void *buffer, size_t *lenp, int conv, int op)
{
        lids_locks_t locks;
        byte   hashcode[RMDsize/8];
        char rmd160sig[170];
.......
                locks.passwd[sizeof(passwd_t)-1]=0; /* We don't take the risk */
                rmd160sig[0]=0;

#ifdef CONFIG_LIDS_ALLOW_SWITCH
                if ((!lids_first_time) || (locks.passwd[0])) {
                        RMD((byte *)locks.passwd,hashcode);
                        memset((char *)locks.passwd,'\0',sizeof(passwd_t));
                        for (i=0; i<RMDsize/8; i++)
                                sprintf(rmd160sig+2*i,"%02x", hashcode[i]);
                }
                if ( ((lids_first_time) && (!locks.passwd[0])) ||
```

```
   ---------->         (!strncmp(rmd160sig,CONFIG_LIDS_RMD160_PASSWD,160)) ) {
#else
                if ((lids_first_time) && (!locks.passwd[0])) {
#endif
                        /* access granted ! */
                        number_failed=0;
                        if (lids_process_flags(locks.flags)) {
                                cap_bset=locks.cap_bset;
                                lids_security_alert("Changed: cap_bset=0x%x lids_flags=0x%x",cap_t(cap_bset),lids_flags);
                        }
                        lids_first_time=0;
                }
........
}
```

After the password checking is ok, the `lids_process_flag()` change the current lids flag with LIDS off and then you can do what you want to do. You can look at the code at `fs/lids.c` of `lids_process_flag` for detail.

7.2.2 switch with LIDS & LIDS_GLOBAL

If you switch the LIDS protection off, you have two choice, firstly, switch off and on other console it is also unprotected by LIDS, secondly, you can switch off only locally, on other console, all the system also protected by LIDS. It can improve security.

The detail impletmetation is in fs/lids of `lids_process_flag()`.

8 Network security in kernel.

With LIDS, you can protect you network with some features shown below.

8.1 Firewall and routing rules protection.

If you host also contains some firwall rules. you can protect them by LIDS. You should turn the `CONFIG_LIDS_ALLOW_CHANGE_ROUTES` on. And you must turn the `CAP_NET_ADMIN` off when sealing the kernel. And then, you can also define allowed programs to change the routing rules.

Let's look at the code of Firewall rules protection. Every request to change the firewall rules will called the kernel function `ip_setsockopt()`.

```
int ip_setsockopt(struct sock *sk, int level, int optname, char *optval, int optlen)
{
      ........
        switch(optname)
        {
        .......
                case IP_FW_DELETE_NUM:
                case IP_FW_INSERT:
```

```
                case IP_FW_FLUSH:
                case IP_FW_ZERO:
                case IP_FW_CHECK:
                case IP_FW_CREATECHAIN:
                case IP_FW_DELETECHAIN:
                case IP_FW_POLICY:
#ifdef CONFIG_LIDS_ALLOW_CHANGE_ROUTES
                        if (!(capable(CAP_NET_ADMIN) || (current->flags & PF_CHROUTES))) {
#else
                        if (!capable(CAP_NET_ADMIN)) {
#endif
#ifdef CONFIG_LIDS
                                lids_security_alert("CAP_NET_ADMIN violation: try to change IP firewall rules with option %d",opt
#endif
                                return -EACCES;
                        }
........
```

From the above code, we can see that if you want to change the firewall rules, you must turn the capabilty `CAP_NET_ADMIN` on and the program you use to change the rules must be marked as `routing_changeable`. The program name shoud be provided when the configurate the kernel.

8.2 Disable Sniffer

The feature is inpletemented as the above changing_route, let's have a look at the code at `net/core/dev.c`.

```
int dev_ioctl(unsigned int cmd, void *arg)
{
........
        switch(cmd)
        {
........
                case SIOCSIFMETRIC:
                case SIOCSIFMTU:
                case SIOCSIFMAP:
                case SIOCSIFHWADDR:
                case SIOCSIFSLAVE:
                case SIOCADDMULTI:
                case SIOCDELMULTI:
                case SIOCSIFHWBROADCAST:
                case SIOCSIFTXQLEN:
                case SIOCSIFNAME:
#ifdef CONFIG_LIDS_ALLOW_CHANGE_ROUTES
                        if (!(capable(CAP_NET_ADMIN) || (current->flags & PF_CHROUTES))) {
#else
                        if (!capable(CAP_NET_ADMIN)) {
#endif
#ifdef CONFIG_LIDS
                                lids_security_alert("CAP_NET_ADMIN violation: ioctl SIOC #%i",cmd);
#endif
```

```
                                        return -EPERM;
        .........
```

You can see that if you want to change the promiscous state needed for sniffer, you must have capability `CAP_NET_ADMIN` on and a corrent program. You should disable the promiscouse state when the network bring up before sealing the kernel and make the `CAP_NET_ADMIN` off when sealing the kernel.

8.3 Port scanner detector in kernel.

8.3.1 Why need a port scanner detector in the kernel.

Since if a port scanner can detect half open scanning, it must be run as a sniffer packet program. If we need a port scanner and we also want the kernel disable promicouse which means that we can not use the sniffer packet programi, the port scanner detector in kernel will be useful.

The main idea for the port scanner is that it use the feature that port scanner alway scanner a rang of ports in a few seconds. And they report the opened port after scan. In this way, the scanner will scan many port which do not listen in the remote machine. In the kernel, we can detect in the following code.

8.3.2 Port scanner detector in kernel

Let's have a look at the tcp port scanner.

```
/* in net/ipv4/tcp_ipv4.c */

int tcp_v4_rcv(struct sk_buff *skb, unsigned short len)
{
        ........
            __skb_queue_tail(&sk->back_log, skb);
            return 0;

no_tcp_socket:

            #ifdef CONFIG_LIDS
            lids_check_scan(skb->nh.iph->saddr,ntohs(th->dest));
            #endif
            tcp_v4_send_reset(skb);

discard_it:

      .........
}
```

The `lids_check_scan()` take two parameters, one for the source address which cause the `no_sock_error`, the other is the port on the machine which the source address want to communicate with but is not open for server.

The main task of the `lids_check_scan()` is to statistic the error number made from the same source. But `lids_check_scan()` do not check if the source address is a port scanner, it let the timer to do it. Now, let's have a look at the `lids_check_scan()`.

```
/* in net/ipv4/lids_check_scan.c */
int lids_check_scan(__u32 addr,__u16 port)
{
...........
        if((p = lids_find_scan(addr)) == NULL) {

                p1 = &lids_scan_head;
                p = (struct lids_scan*)kmalloc(sizeof(struct lids_scan),GFP_ATOMIC);
                if(p == NULL ) {
                        return -1;
                }
                while((p1->next)!=NULL)p1=p1->next;

                /* init the structure. */
                p1->next = p;
                spin_unlock(p->lock);
                p->next = NULL;
                p->addr = addr;
                p->counter = 0;
                p->lower_counter = 0;
                p->create_time = current_time;
                /* init a timer to do the detect thing */
                init_timer(&(p->timer));
                p->timer.expires = LIDS_SCAN_TIMEOUT + current_time;
                p->timer.data = (unsigned long) p;
                p->timer.function = lids_proceed_scan ;
                add_timer(&(p->timer));
        }
/* add the counter when hit */
        spin_lock(p->lock);
        (p->counter)++;
/* we here defined the port < 1024 and > 1024 */
        if(port < 1024)
                (p->lower_counter)++;
        spin_unlock(p->lock);
        return 0;
}
```

From the above code, we can see the the function just maintant the list , so it is faster. In order to prevent the DoS attack on the kmalloc(), we also need to limited the detected list. It may be a fault in this code, but since the timer function – `lids_proceed_scan` update the list very fast – every 3 second once. So the DoS attack is every difficult to make the kernel confuse about which is the true scaner source.

9 Intrusion Response System

When we detect someone or some program violate the rules, we must respone to the action. In the current LIDS, we can log the information via klog with the feature of anti-flood loging. We also have the feature to hang up the console which the misbehavious user on. In the future, we will add more response system to the LIDS, not only in the kernel, but also in user space.

9.1 Allow logging in a security way

Most of the code and ideas is from solar desinger's Linux OpenWall project. Thanks solar.

With the traditional logging in the kernel, we use the printk(KERN_XXX) every time we need to print a message to the console. But it is every easily used by other misbehivious to make a DoS attack to the kernel. He can make the kernel running the printk very frequence and then make the filesystem out of free space. With the current security logging facility, we can just use the security_alert() in the kernel, let the function do the other anti-DoS job.

You can have a look at the source at `include/linux/kernel.h`.

9.2 hangup the console.

This feature use with the security_log to make the user who violate the rules defined in LIDS quickly being hangup. He have to relogin the system to continus. But what he have done have log into the system log file or also send to the adminstration by the mail tools developing by Boidi.

9.3 Notify the Administrator by mail and pager.

This feature is developed by Boidi now. With the tools, we can easy know what is wrong with the system, we can response to the intrusion every quickly.

It has been released in lids-0.9pre1, it create a kernel thread to do the communicate things. For details, look at the kernel source for LIDS.

10 Thanks .

First of all, I want to thank Kate Lee who always encourage me to continue writing the document and help me correct many errants in this document.This document is dedicate to her.

Many thanks must go to Biodi Phillipe and Christophe Long who have make many contribut to the LIDS project.

Many thanks also must go to all the LIDS users, without your encourge and feedback and those great ideas, the project can not develop so well.

Finally, I want to thanks my supervisor - Prof. Suo Bai and Ph.d Dongbo Bu, without their helps, I can not even start the LIDS.