Security at Kernel Level
LIDS

Philippe Biondi
<biondi@cartel-securite.fr>

May 30, 2002
Outline

- Why?
  - Context
  - A new security model
  - Conclusion

- How?
  - Taxonomy of action pathes
  - Defending kernel space
  - Filtering in kernel space

- Implementations
  - LIDS
  - Other projects
  - LSM
Outline

Why ?
- Context
- A new security model
- Conclusion

How ?
- Taxonomy of action pathes
- Defending kernel space
- Filtering in kernel space

Implementations
- LIDS
- Other projects
- LSM
We would like to be protected from

- Fun/hack/defacing
- Tampering
- Resources stealing
- Data stealing
- Destroying
- DoS
- ...
Thus we must ensure

- Confidentiality
- Integrity
- Availability

What do we do to ensure that?

- We define a set of rules describing the way we handle, protect and distribute information
  - This is called a security policy
To enforce our security policy, we will use some security software

- Tripwire, AIDE, for integrity checks
- SSH, SSL, IP-SEC, for confidentiality
- Passwords, secure badges, biometric access controls
- ...

Can we trust them? Do they live in a trusted place?
The mice and the cookies

- Facts:
  - We have some cookies in a house
  - We want to prevent the mice from eating the cookies
The mice and the cookies

- Solution 1: we protect the house
  - too many variables to cope with (lots of windows, holes, ...)
  - we can’t know all the holes to lock them.
  - we can’t be sure there weren’t any mice before we closed the holes

  *I won’t bet I’ll eat cookies tomorrow.*

- Solution 2: we put the cookies in a metal box
  - we can grasp the entire problem
  - we can “audit” the box
  - the cookies don’t care whether mice can break into the house

  *I’ll bet I’ll eat cookies tomorrow.*
Usual security model

硬件
内核空间
用户空间
tripwire
sendmail
ssh
trusted
Usual security model
Kernel security model

- hardware
- kernel space
- user space
- tripwire
- sendmail
- ssh

trusted
untrusted
To use this model, we must patch the kernel for it to

- protect itself
  - trusted kernel space
- protect other programs/data related to/involved in the security policy
## Why?
- Context
- A new security model
- Conclusion

## How?
- Taxonomy of action paths
- Defending kernel space
- Filtering in kernel space

## Implementations
- LIDS
- Other projects
- LSM
- Bugless interfaces
  - network stack, kbd input, ...
  - system calls

- Defence
  - /dev/mem, /dev/kmem ...
  - create_module(), init_module(), ...

- Filtering
  - Queries to reach a storage device or PROMs, FPGAs, ...
  - Queries to reach another process' memory
Is the bugless interface hypothesis ok?

- Protected mode mechanisms $\implies$ harder to do programming faults (IMHO) (bugs are still possible, race conditions for ex.)

```c
linux/drivers/char/rtc.c

static int rtc_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
                      unsigned long arg)
{
    unsigned long flags;
    struct rtc_time wtime;

    switch (cmd) {
        [...]
        case RTC_ALM_SET: /* Store a time into the alarm */
        {
            unsigned char hrs, min, sec;
            struct rtc_time alm_tm;

            if (copy_from_user(&alm_tm, (struct rtc_time*)arg, sizeof(struct rtc_time)))
                return -EFAULT;
```
How to protect kernel space against a user space intruder?
Block everything from user space that can affect kernel space.

- Attacks can come through:
  - system calls
  - devices files
  - procfs

- Few entry points, opened by the kernel
  - /dev/mem, /dev/kmem
  - /dev/port, ioperm and iopl
  - create_module(), init_module(),...
  - reboot()
/dev/mem, /dev/kmem and /dev/port protection:

```c
static int open_port(struct inode * inode, 
                     struct file * filp) 
{
    return capable(CAP_SYS_RAWIO) ? 0 : -EPERM;
}
```
Module insertion control:

```c
asmlinkage unsigned long
sys_create_module(const char *name_user, size_t size)
{
    char *name;
    long namelen, error;
    struct module *mod;

    if (!capable(CAP_SYS_MODULE))
        return -EPERM;

    [...]
```
What must we protect?

- What is in memory
  - Processes (memory tapering, IPC, network communications, ...)
  - Kernel configuration (firewall rules, etc.)

- What is on disks or tapes
  - Files
  - Metadata (filesystems, partition tables, ...), boot loaders, ...

- Hardware
  - Devices (ioctl, raw access, ...)
  - EPROMs, configurable hardware, ...
How to protect that?

- Queries are done only via the kernel
- System calls, sysctls and devices drivers are a place of choice for controlling accesses
  - We have to modify their behaviour consistently to be able to enforce a complete security policy.
A good way is to use a modular architecture to control kernel calls: there will be

- An enforcer component
- A decider component
  - Lots of access control policies (DAC, MAC, ACL, RBAC, IBAC, ...)

Diagram:

- App
- Kernel call
- Enforcer component
- Decider component
How to add the enforcer code to the kernel calls?

- kernel call interception
- kernel call modification

ex: system call anatomy:
Syscall interception example : Medusa DS9

```c
linux/arch/i386/kernel/entry.S

[...]

GET_CURRENT(%ebx)
cmpl $(NR_syscalls),%eax
jae badsys

#ifdef CONFIG_MEDUSA_SYSCALL
/* cannot change: eax=syscall, ebx=current */
btl %eax,med_syscall(%ebx)
jnc 1f
pushl %ebx
pushl %eax
call SYMBOL_NAME(medusa_syscall_watch)
cmpl $1, %eax
popl %eax
popl %ebx
jc 3f
jne 2f
1:
#endif

testb $0x20,flags(%ebx)    # PF_TRACESYS
jne tracesys

[...]
```
Syscall interception advantages

- general system
- low cost patch

Drawbacks

- kind of duplication of every syscall
- need to know and interpret parameters for each different syscall
- architecture dependent
Syscall modification example: LIDS
linux/fs/open.c

```c
asmlinkage long sys_utime(char * filename, struct utimbuf * times)
{
    int error;
    struct nameidata nd;
    struct inode * inode;
    struct iattr newattrs;

    error = user_path_walk(filename, &nd);
    if (error)
        goto out;
    inode = nd.dentry->d_inode;

    error = -EROFS;
    if (IS_RDONLY(inode))
        goto dput_and_out;

    #ifdef CONFIG_LIDS
        if(lids_load && lids_local_load) {
            if (lids_check_base(nd.dentry, LIDS_WRITE)) {
                lids_security_alert("Try to change utime of \\
                                %s", filename);
                goto dput_and_out;
            }
        }
    #endif

    /* Don’t worry, the checks are done in inode_change_ok() */
    newattrs.ia_valid = ATTR_CTIME | ATTR_MTIME | ATTR_ATIME;
    if (times) {
```
- Syscall modification advantages
  - Syscall parameters already interpreted and checked
  - Great tuning power. We can alter the part of the syscall we want.

- Drawbacks
  - Lot of the 200+ syscalls must be altered
To be out soon in the kernel: LSM
linux/kernel/module.c

sys_create_module(const char *name_user, size_t size)
{
    char *name;
    long namelen, error;
    struct module *mod;
    unsigned long flags;

    if (!capable(CAP_SYS_MODULE))
        return -EPERM;
    lock_kernel();
    if ((namelen = get_mod_name(name_user, &name)) < 0) {
        error = namelen;
        goto err0;
    }
    if (size < sizeof(struct module)+namelen) {
        error = -EINVAL;
        goto err1;
    }
    if (find_module(name) != NULL) {
        error = -EEXIST;
        goto err1;
    }
    /* check that we have permission to do this */
    error = security_ops->module_ops->create_module(name, size);
    if (error)
        goto err1;
}

C
ARTEL
SÉCURITÉ — Philippe Biondi
Outline

Implementations

- Why?
  - Context
  - A new security model
  - Conclusion

- How?
  - Taxonomy of action pathes
  - Defending kernel space
  - Filtering in kernel space

- Implementations
  - LIDS
  - Other projects
  - LSM
Linux Intrusion Detection System

- Self-protection
- Processes protection
- Files protection
- Online administration
- Special (controversial) features
  - Dedicated mailer in the kernel
  - Kind of portscan detector in the kernel
Self-protection

- Modules insertion/deletion, `/dev/mem`, ..., `ioperm/iopl`, ... filtered

- Boot process protected
  - Can forbid the execution of non-protected programs (not flawless)

- Sealing mechanism
  - `fsck` or `insmod` can run when booting
  - No human intervention is needed to seal the protection
  - After the seal, we are in the working state. Everything is locked
Processes protection

- Rely on the Linux capabilities bounding set
  - Hardware protection
  - Processes privacy (ptrace, promiscuous mode, ... can be forbidden)
  - Network administration locked, ...

- Daemons can be made unkillable

- Processes can be made invisible

- Processes can be granted capabilities

```
lidsconf -A -s /usr/sbin/sshd \ 
  -o CAP_NET_BIND_SERVICE 22-22 -j GRANT
```
Files protection

- MAC-like approach:
  
  ```bash
  lidsadm -A -s /usr/sbin/httpd \\
  -o /home/httpd -j READ
  ```

- Files identified by VFS device/inode ⇒ works on every fs
Online administration

- LIDS can be disabled globally
- LIDS can be reconfigured on the fly
- LIDS can be disabled only for a shell and its children
Special features

- Mailer in the kernel
  - Can make a network connection (TCP or UDP)
  - Can send a scriptable session (mail, syslog, ...)
  - Does not rely on anything in user space

- Scan detector in the kernel
  - Kind of interrupt driven ⇒ no load at all
  - Does not need the promiscuous mode
  - Works on all interfaces at the same time
  - Detect only connect/syn scans
  - Detect only what reach the TCP or UDP stack
Other projects

- LIDS
- Medusa DS9
- RSBAC
- LoMaC
- SE Linux
- . . .
Linux Security Modules: to be included in 2.5

Kernel Summit 2001: Linus decides that Linux should support security enhancements

LSM patch is a set of hooks in the kernel system calls

- Linux kernel provides the enforcer component

Modular enough for the decider component to become a LKM
That’s all folks. Thanks for your attention.

You can reach me at <phil@lids.org>

These slides are available at
http://www.cartel-securite.fr/pbiondi/