Security at Kernel Level

Philippe Biondi
<biondi@cartel-securite.fr>

—

Libre Software Meeting
July 10, 2002
Why?
- Context
- A new security model
- Conclusion

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

Implementations
- LIDS
- Existing projects
- LSM
Outline

Why ?
- Context
- A new security model
- Conclusion

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

Implementations
- LIDS
- Existing 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, bsign, ... for integrity checks
- SSH, SSL, IP-SEC, PGP, ... for confidentiality
- Passwords, secure badges, biometric access controls, ... for authentication
- ...

Can we trust them? Do they work 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

- hardware
- kernel space
- user space
- tripwire
- sendmail
- ssh
- trusted
Usual security model

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

Why?
Context | New model | Conclusion
Kernel security model

hardware

trusted

untrusted

kernel space

tripwire

sendmail

ssh

user space
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
Outline

Why?
- Context
- A new security model
- Conclusion

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

Implementations
- LIDS
- Existing projects
- LSM
How?

Taxonomy | Defence | Filtering

Targets

- human
- physical security
  - action vehicle
  - storage
  - PROM, FPGA, ...
- kernel
- MMU
  - application
  - application
  - application

CARTEL SÉCURITÉ — Philippe Biondi
Targeting storage or PROM with direct access to the box

- **human**
- **physical security**
- **action vehicle**
- **storage**
- **PROM, FPGA,...**
- **kernel**
- **MMU**
- **application**
Targeting an application accessible with keyboard, network, ...
Targeting storage or PROM through an accessible application
Targeting an unaccessible application through an accessible one

How?

Taxonomy | Defence | Filtering
Targeting kernel directly or through an accessible application

Human

Physical security

Action vehicle

Storage

PROM, FPGA,...

Kernel

MMU

Application

Application

Application
- Bugless interfaces
  - network stack, kbd input, …
  - kernel 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
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()
How? Taxonomy | Defence | Filtering

/how/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 tampering, 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, …)
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
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  
[...]

CARRIER  —  Philippe Biondi
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

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

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

Implementations
- LIDS
- Existing 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:

  lidsadm -A -s /usr/sbin/httpd \\n  -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 \( \Rightarrow \) 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
LIDS general architecture

- **Boot stuff**
  - Kernel image
  - LIDS AC data
  - init, rc, daemons
- **Kernel image**
  - lidsadm
  - LIDS AC data
- **procfs stuff**
  - init code
  - AC data
- **init code**
  - decider component
  - portscan detector
- **syscalls**
  - enforcer component
  - Logging stuff
  - Kernel Mailer
- **processes**
  - syslog
- **syslog**
Other projects

- Openwall
- GrSecurity
- Medusa DS9
- RSBAC
- LoMaC
- SE Linux
- ...

CARTEL SÉCURITÉ — Philippe Biondi
Openwall: Collection of security-related features for the Linux kernel.

- Non-executable user stack area
- Restricted links in /tmp
- Restricted FIFOs in /tmp
- Restricted /proc
- Special handling of fd 0, 1, and 2
- Enforce RLIMIT_NPROC on execve
GrSecurity : General Security for Linux

- Kernel hardening from Openwall
- ACL system
**Medusa DS9**: Extending the standard Linux (Unix) security architecture with a user-space authorization server.

- **layer 1**
  - Hooks in the original kernel code

- **layer 2**
  - kernel space code
  - called from hooks.
  - do basic permission checks
  - check for cached permissions
  - call the communication layer if necessary

- **layer 3**
  - communication layer
  - communicate with a user space daemon
- User space daemon
  - decider component
- Miscellaneous
  - syscall interception
  - can force code to be executed after a syscall
RSBAC: Rule Set Based Access Control

- It is based on the Generalized Framework for Access Control (GFAC)
- All security relevant system calls are extended by security enforcement code.
- Different access control policies implemented as kernel modules
  - MAC, ACL, RC (role control), FC (Functional Control), MS (Malware Scan), ...
LOMAC: Low Water-Mark Integrity

- **Initialization**
  - Some specified directories ($B$) are high
  - Other directories ($D$) and sockets ($E$) are low

- **Execution**
  - Processes created from $B$ are high
  - Processes created from $D$ are low

- **Reading**
  - $A$ can read $B$. $C$ can read $D$ or $E$
  - $C$ can’t read $B$
  - if $A$ reads $D$ or $E$, $A$ goes into the low level

- ...
**SE Linux**: NSA’s Security Enhanced Linux

- Based on the Flask architecture  
  (Flexible architecture security kernel)
- Enforcer / decider components
- Pays a lot of attention to the change of the access control policy  
  (revocation)
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 syscalls

- Linux kernel provide 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/