Security at Kernel Level

(again)

Philippe Biondi

<phil@secdev.org>

<philippe.biondi@arche.fr>

—

Linux Kongress

October 15, 2003

- Why ?
  - Context
  - A new security model
  - Conclusion
- How ?
  - Taxonomy of action paths
  - Defending kernel space
  - Filtering in kernel space
- Implementations
  - Existing projects
  - LSM
  - LSM code example

- Why ?
  - Context
  - A new security model
  - Conclusion
- How ?
  - Taxonomy of action paths
  - Defending kernel space
  - Filtering in kernel space
- Implementations
  - Existing projects
  - LSM
  - LSM code example

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, debsum, . . . 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

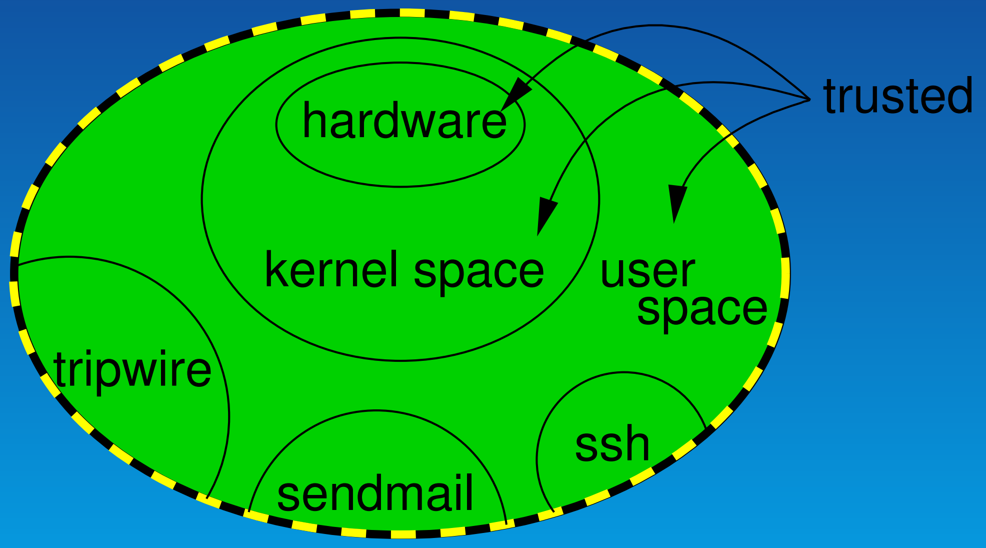
- Solution 1 : we protect the house
  - too many variables to cope with (lots of windows, holes, . . .)
  - we can't know all the holes to lock them.
  - we can't be sure there weren't any mice before we closed the holes

  I won't bet I'll eat cookies tomorrow.

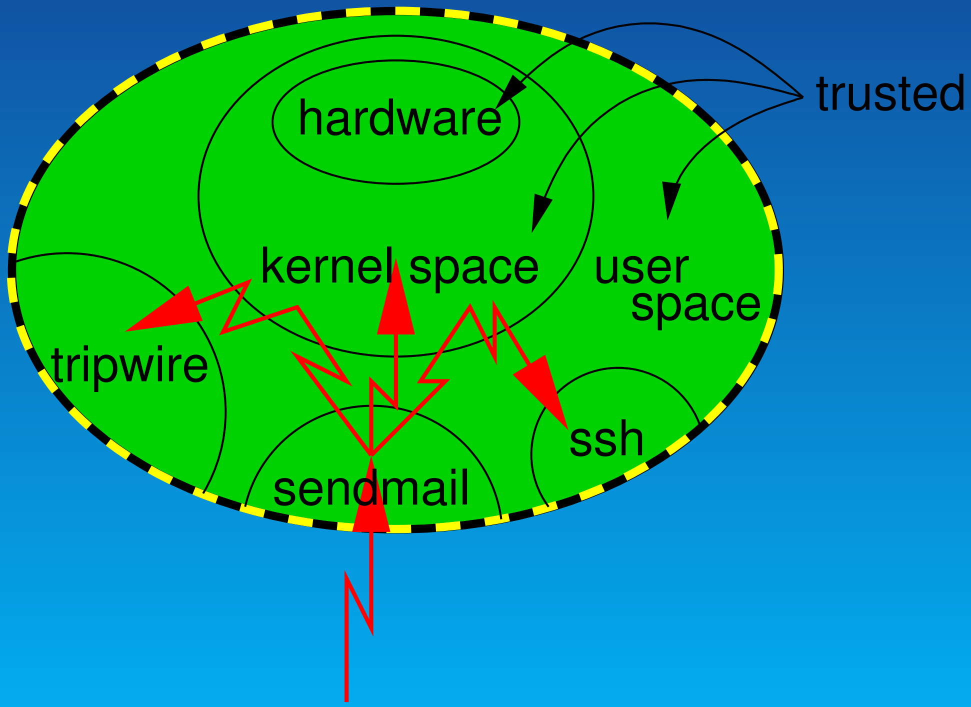
- Solution 2 : we put the cookies in a metal box
  - we can grasp the entire problem
  - we can "audit" the box
  - the cookies don't care whether mice can break into the house

  I'll bet I'll eat cookies tomorrow.

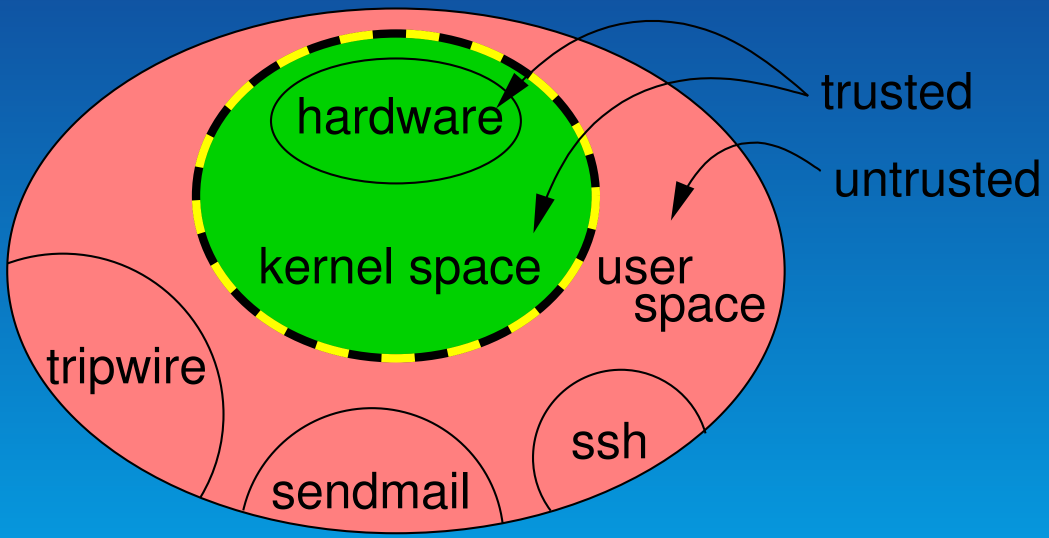
Usual security model



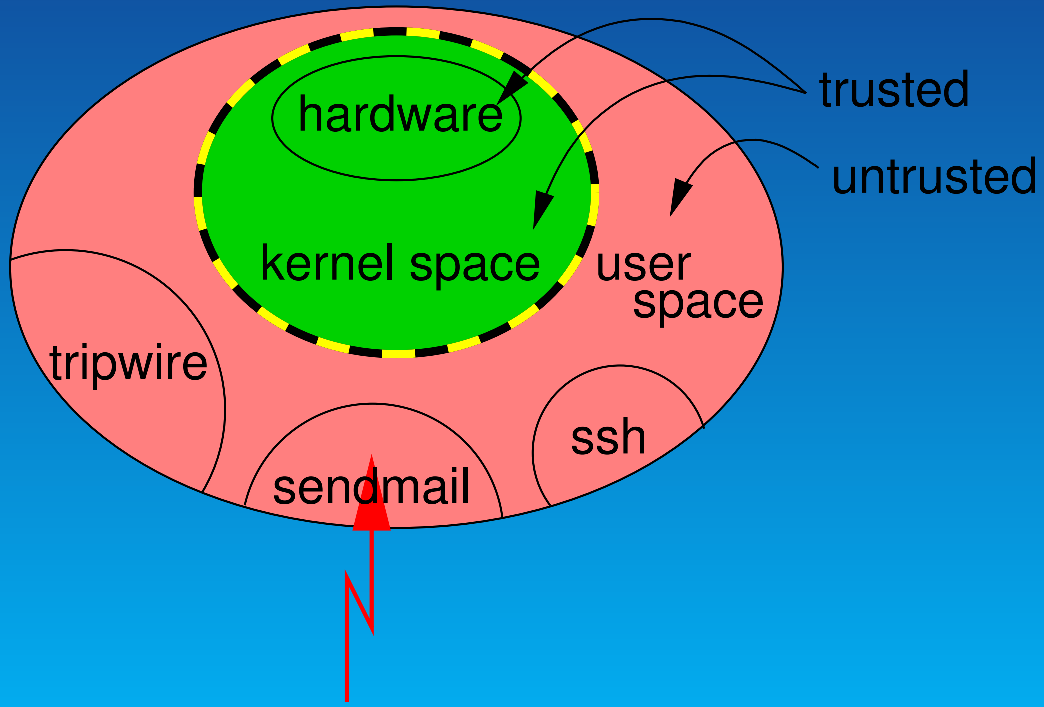
Usual security model



Kernel security model



Kernel security model

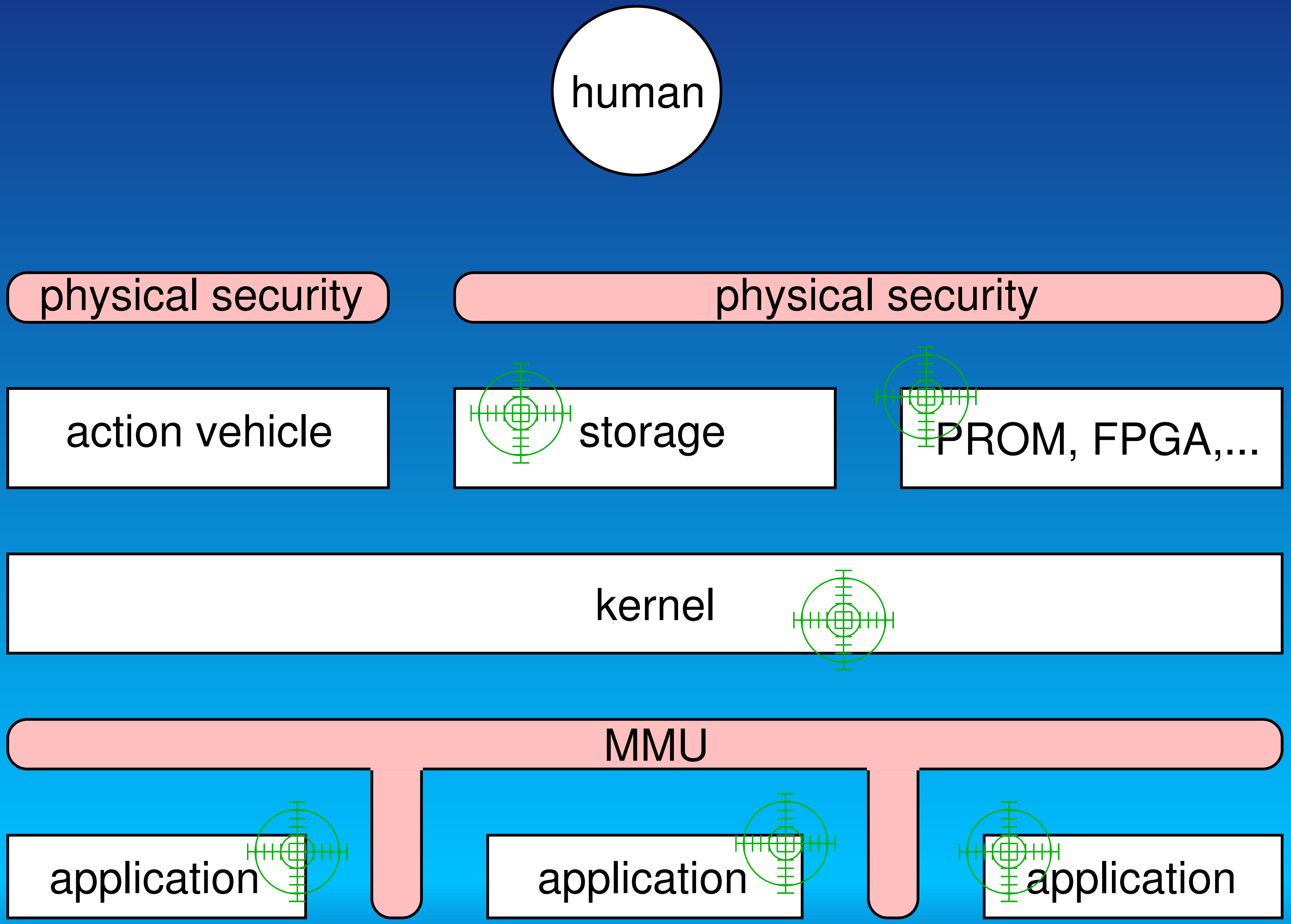


To use this model, we must patch the kernel for it to

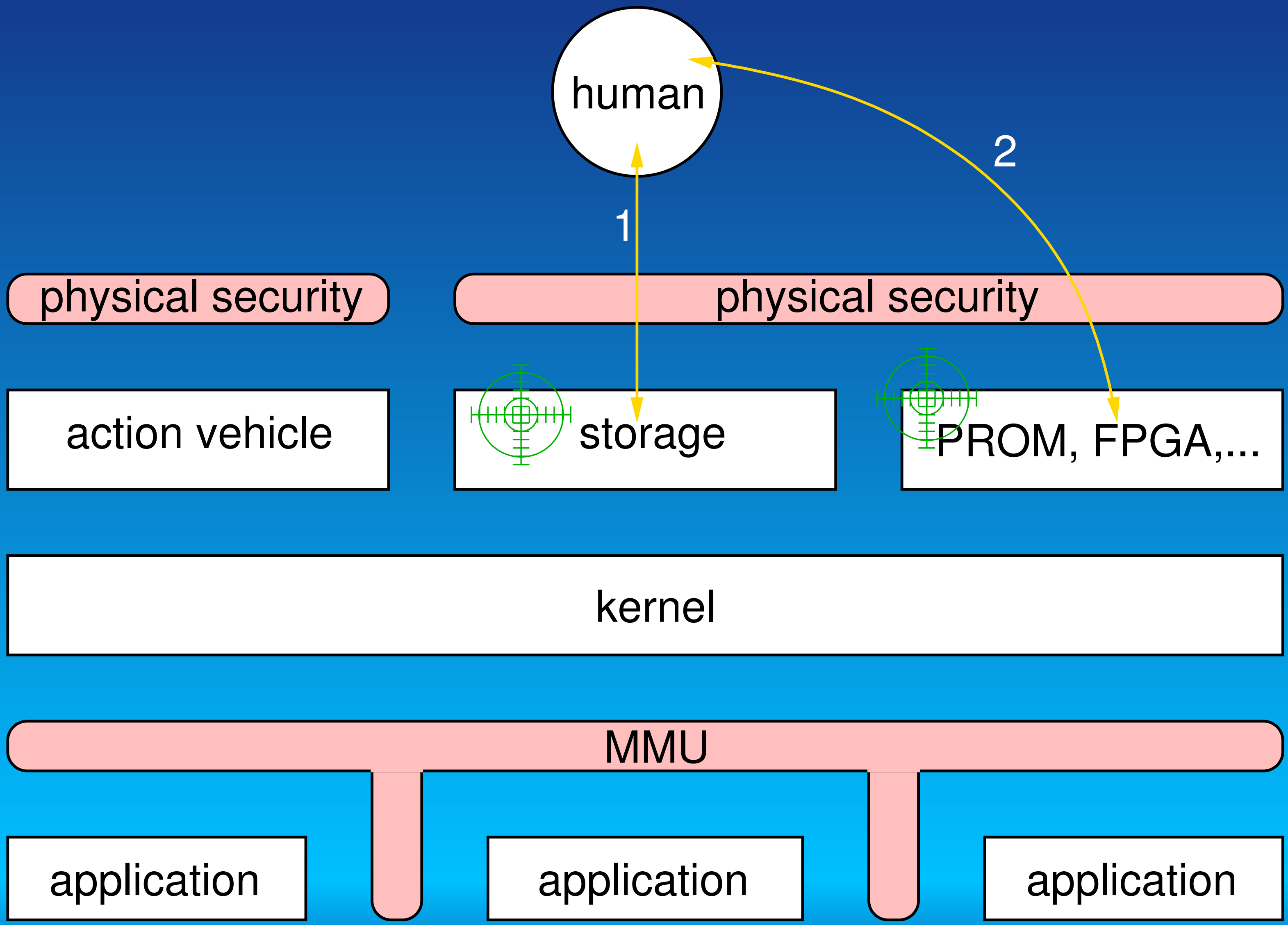
- protect itself
  - ➥ trusted kernel space
- protect other programs/data related to/involved in the security policy

- Why ?
  - Context
  - A new security model
  - Conclusion
- How ?
  - Taxonomy of action paths
  - Defending kernel space
  - Filtering in kernel space
- Implementations
  - Existing projects
  - LSM
  - LSM code example

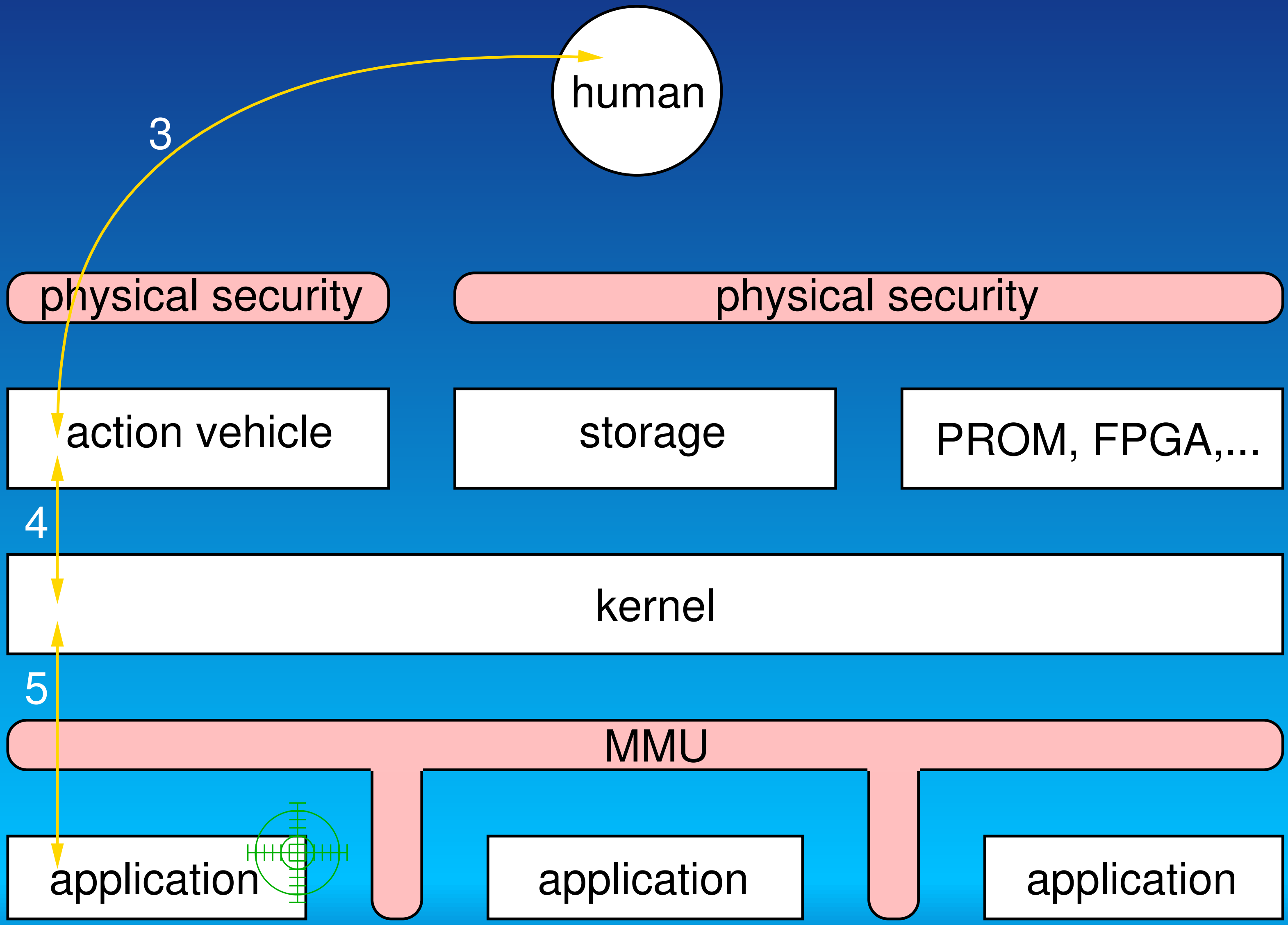
Targets


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

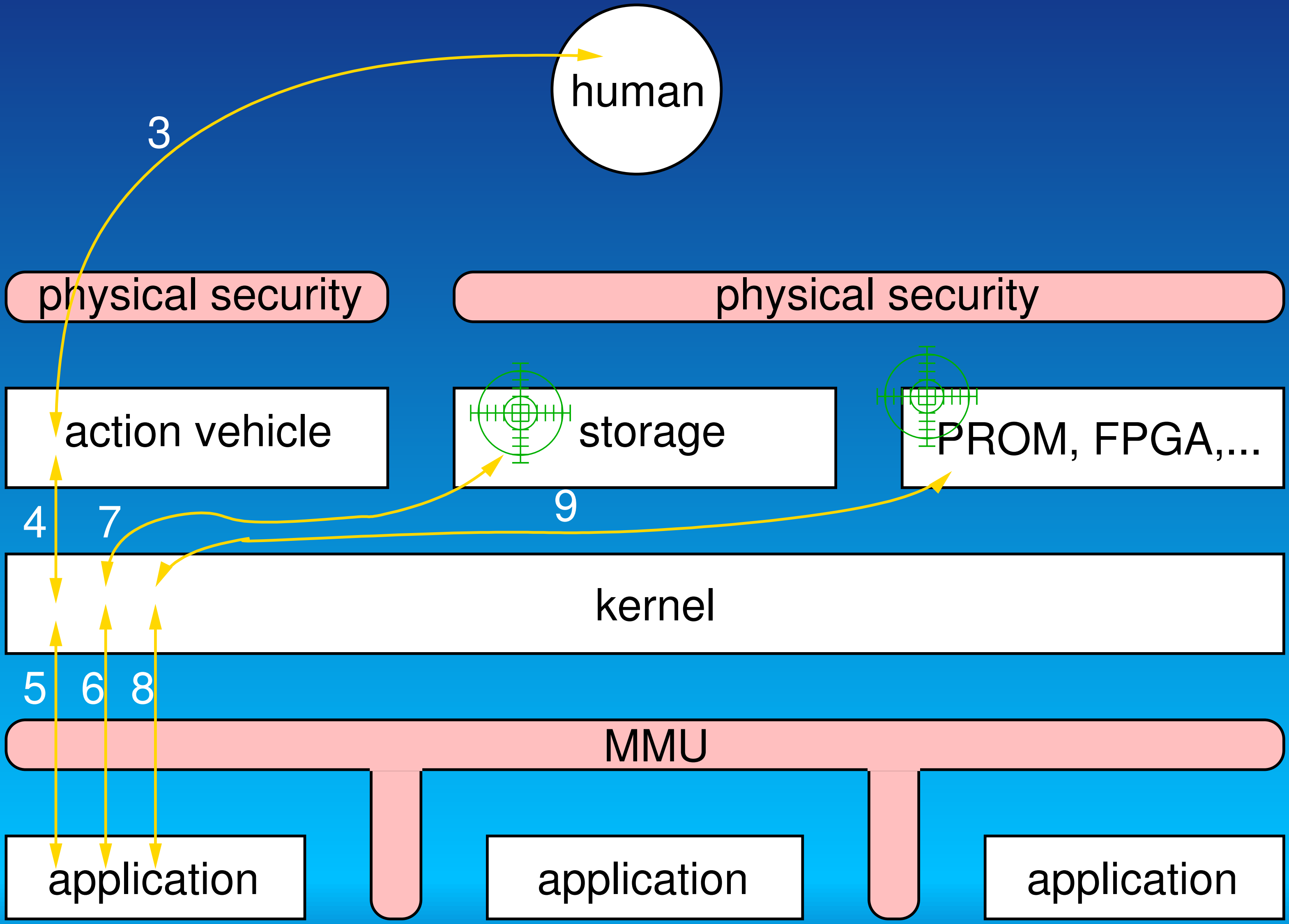
Targeting storage or PROM with direct access to the box



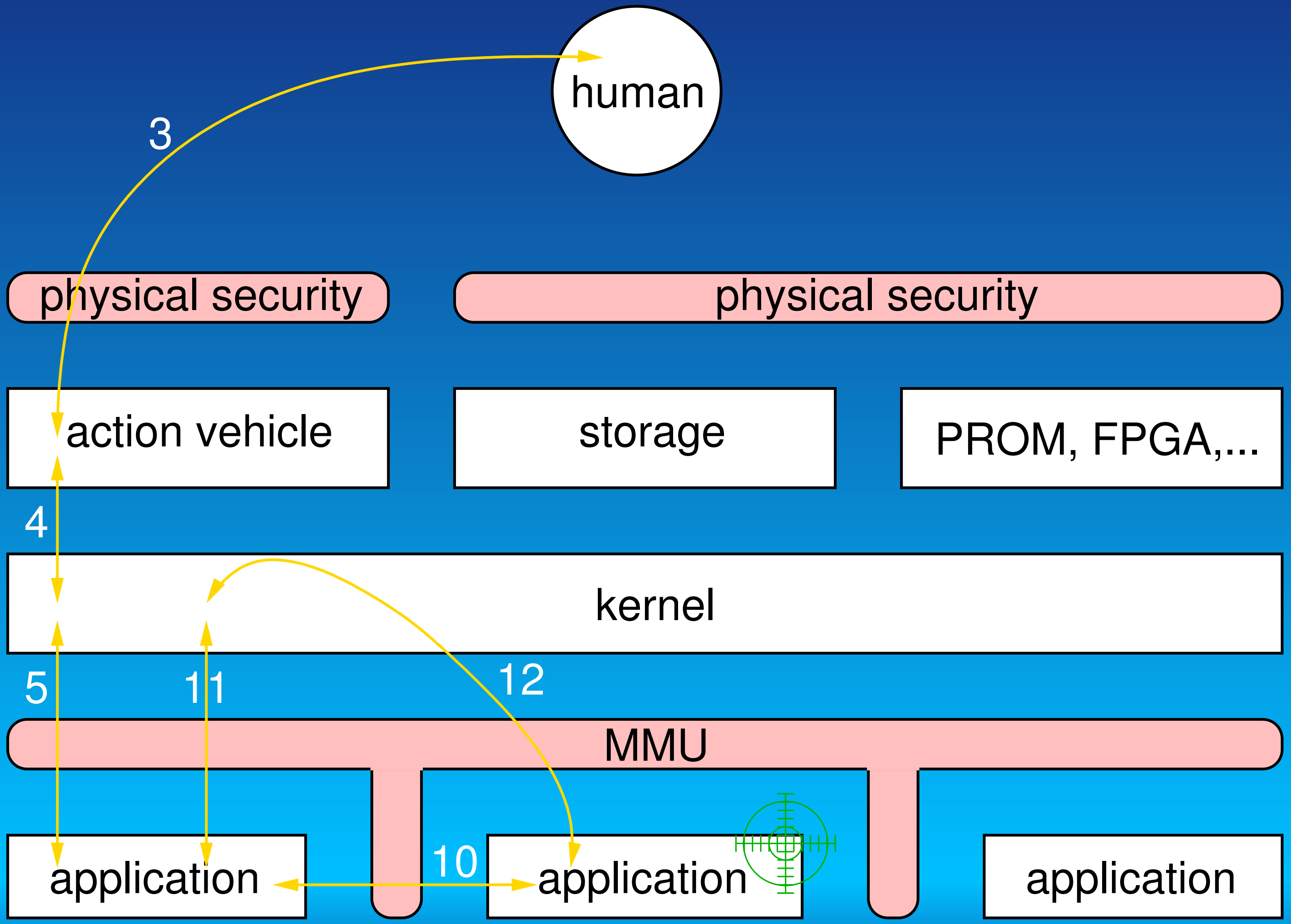
Targeting an application accessible with keyboard, network, . . .



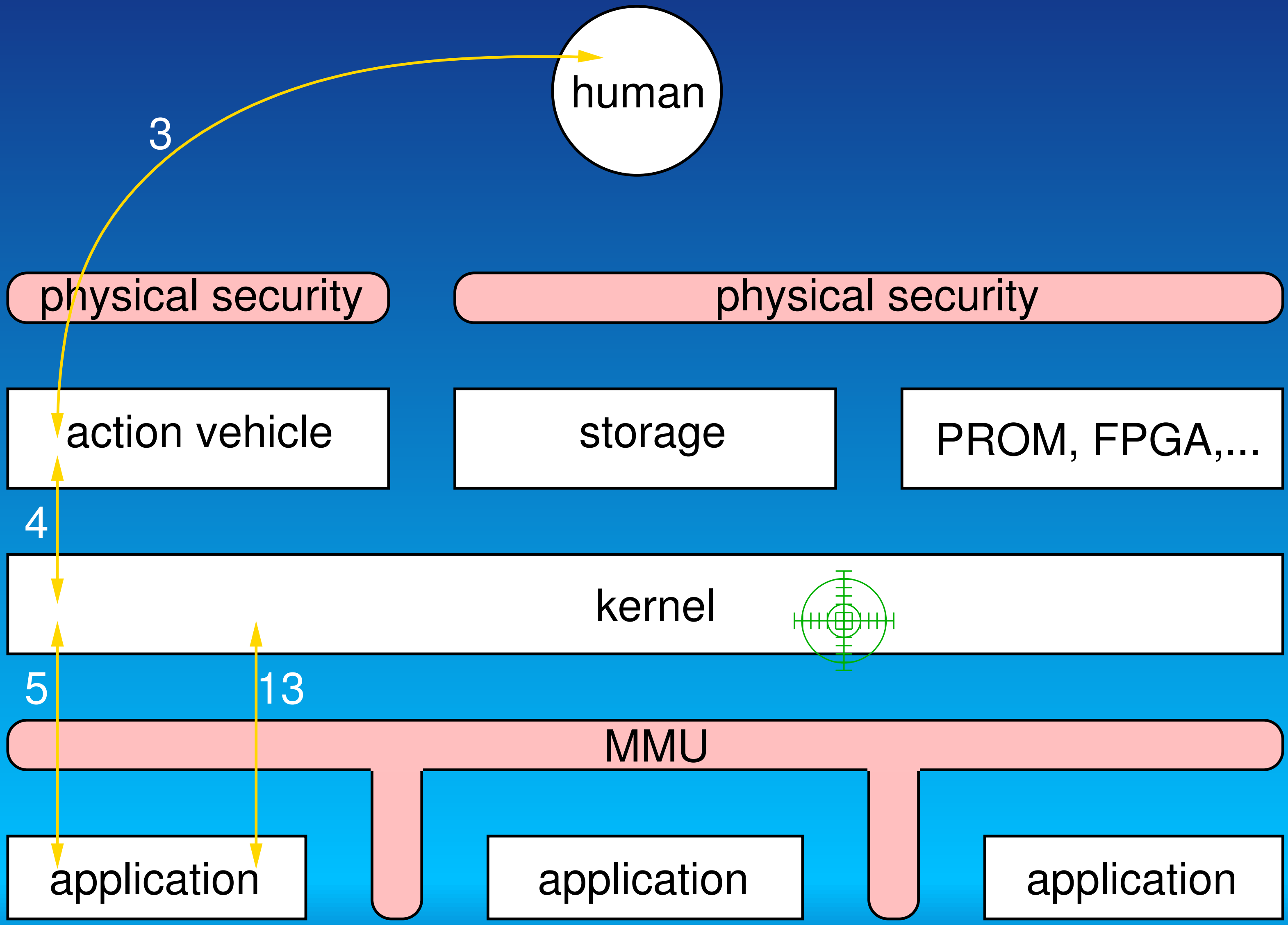
Targeting storage or PROM through an accessible application



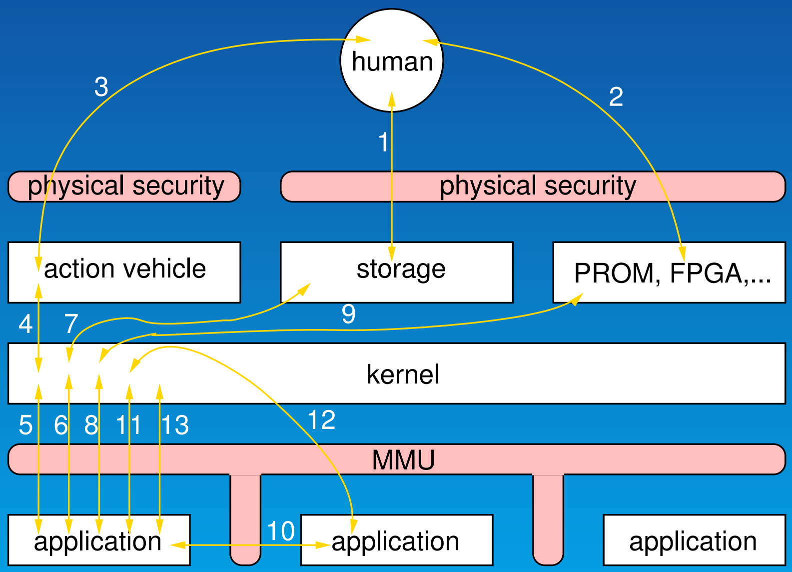
Targeting an unaccessible application through an accessible one



Targeting kernel directly or through an accessible 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



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 :

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

- Module insertion control :

```
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 ?

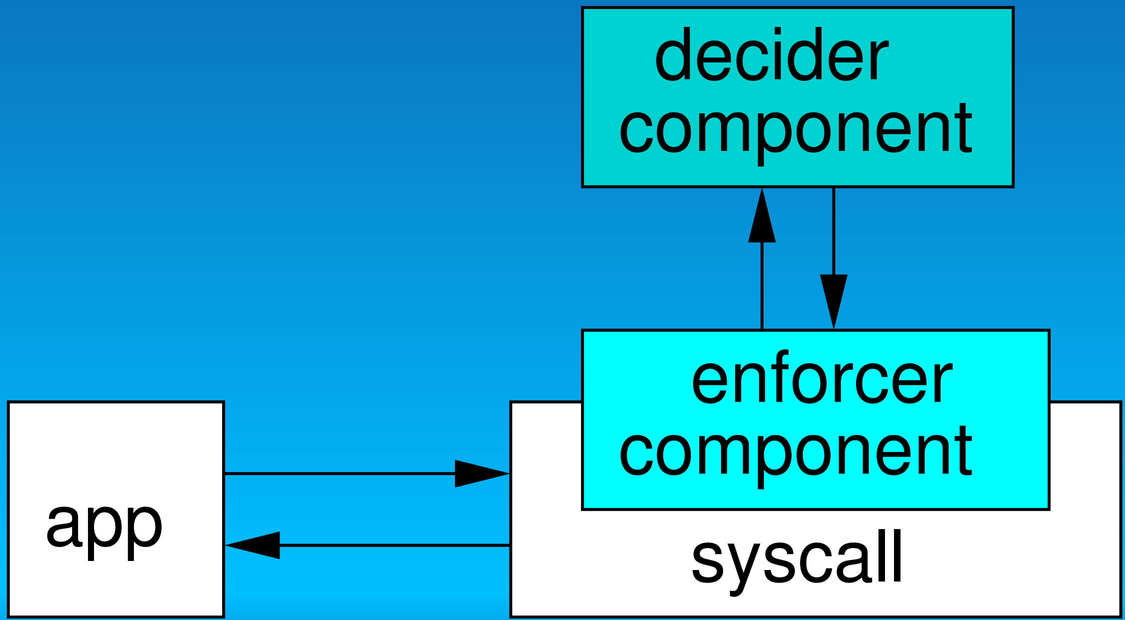
- What is in memory
  - Processes (memory tampering, IPC, network communications, . . .)
  - Kernel configuration (firewall rules, etc.)
- What is on disks or tapes
  - Files
  - Metadata (filesystems, partition tables, . . .), boot loaders, . . .
- Hardware
  - Devices (ioctl, raw access, . . .)
  - EPROMs, configurable hardware, . . .

How to protect that ?

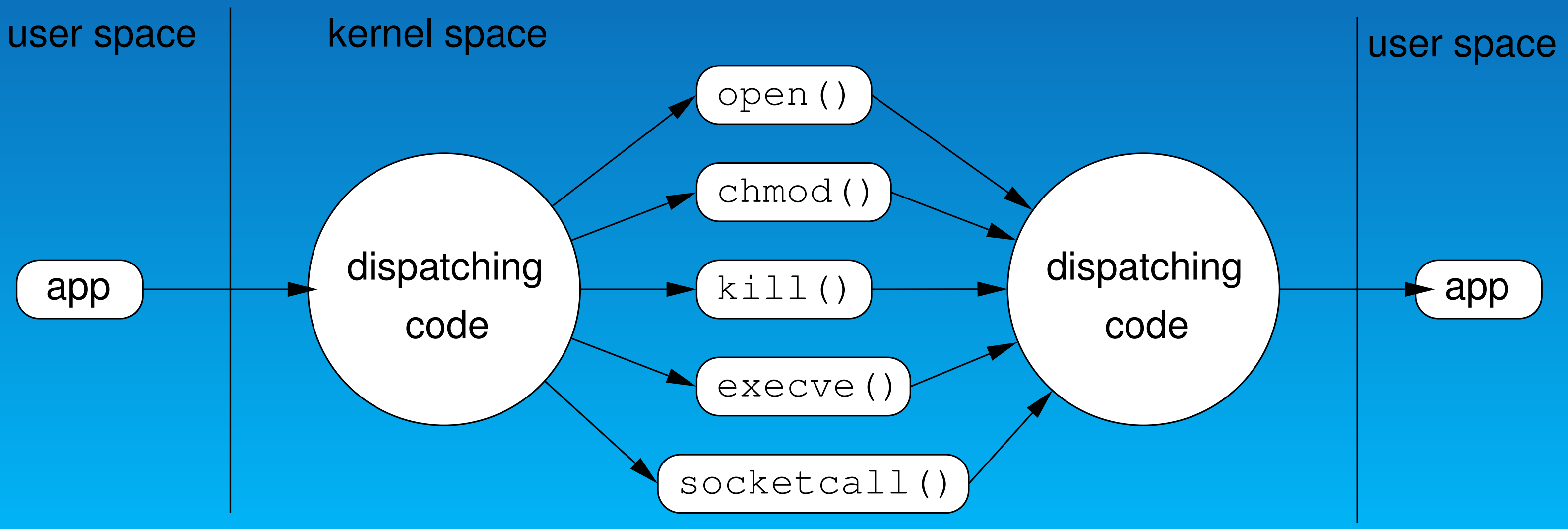
- Queries are done only via the kernel
- System calls, sysctls and devices drivers are a place of choice for controlling accesses
  - ➥ We have to modify their behaviour consistently to be able to enforce a complete security policy.

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
[...]
```

- Syscall interception advantages
  - generic 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) {
```

Linux Security Module
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->create_module(name, size);
       if (error)
               goto err1;
```

- 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

- Why ?
  - Context
  - A new security model
  - Conclusion
- How ?
  - Taxonomy of action paths
  - Defending kernel space
  - Filtering in kernel space
- Implementations
  - Existing projects
  - LSM
  - LSM code example

Other projects

- pH
- Openwall
- PaX
- GrSecurity
- Medusa DS9
- Systrace
- RSBAC
- LIDS
- LoMaC
- SE Linux

pH :

- HIDS with ICE (counter-measures)
  - learns order of syscalls for given processes
  - detects any deviance from the learned model
  - the more it deviates, the more syscalls are slowed dow

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

PaX :

- hardening patch against buffer overflow exploitations, format strings,. . .
  - Non-executable stack
  - full Address Space Layout Randomization (ASLR)

GrSecurity : General Security for Linux

- Include PaX
- Include kernel hardening features from Openwall
- ACL system
- PID, RPC XID, TCP source ports randomization
- Auditing functionnalities

Medusa DS9 : Extending the standard Linux (Unix) security architecture with a user-space authorization server

- Uses system call interception
- can force code to be executed after a syscall
- User space autorization server (decider component)
  - ➥ easier to write lots of access control policies
  - ➥ does not fit in the everything-essential-in-kernel-space model

Systrace :

- *BSD and Linux kernels
- System call inteception
- by-process system call control
  - which syscalls are permitted
  - which parameters are permitted for a given syscall
- per-syscall privilege elevation
- automatic policy generation

RSBAC : Rule Set Based Access Control

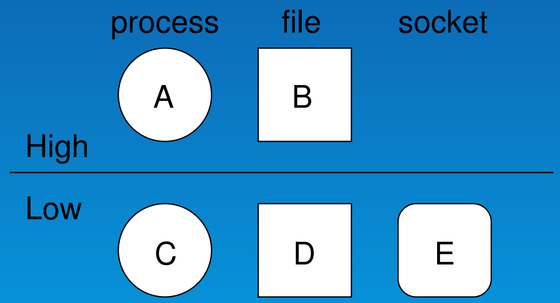
- It is based on the Generalized Framework for Access Control (GFAC)
- All security relevant system calls are extended by security enforcement code.
- Different access control policies implemented as kernel modules
  - MAC, ACL, RC (role control), FC (Functional Control), MS (Malware Scan), . . .

LIDS :

- bottom-up developpment (no underlying theory)
- everything needed for LIDS to work is in the kernel
- sealing mechanism at boot time
- can work over LSM framework

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)
- Now works on the LSM framework
- Enforcer / decider components
- Pays a lot of attention to the change of the access control policy (revocation)

- Kernel Summit 2001 : Linus decides that linux should support security enhancements
- Many projects already do that. Linus does not want to choose one.
  - ➥ The part included in the kernel should be a framework
  - ➥ should be modular enough for the decider component to become a LKM
- Linux Security Modules : now included in 2.6 !

What are LSM

- LSM add to the kernel
  - Security data fields to some kernel data structures
  - Security data management hooks
  - Hooks at critical points in the kernel code (Enforcer component)

Security fields :

```
struct task_struct {
        volatile long state;
[...]
        void *security;
[...]
}
```

Security data management hooks (1/2):

```
int (*bprm_alloc_security) (struct linux_binprm * bprm);
void (*bprm_free_security) (struct linux_binprm * bprm);
int (*sb_alloc_security) (struct super_block * sb);
void (*sb_free_security) (struct super_block * sb);
int (*inode_alloc_security) (struct inode *inode);
void (*inode_free_security) (struct inode *inode);
int (*file_alloc_security) (struct file * file);
void (*file_free_security) (struct file * file);
int (*task_alloc_security) (struct task_struct * p);
void (*task_free_security) (struct task_struct * p);
int (*msg_msg_alloc_security) (struct msg_msg * msg);
void (*msg_msg_free_security) (struct msg_msg * msg);
int (*msg_queue_alloc_security) (struct msg_queue * msq);
void (*msg_queue_free_security) (struct msg_queue * msq);
int (*shm_alloc_security) (struct shmid_kernel * shp);
void (*shm_free_security) (struct shmid_kernel * shp);
int (*sem_alloc_security) (struct sem_array * sma);
void (*sem_free_security) (struct sem_array * sma);
```

Security data management hooks (2/2):

- inode creation enforcement

```
int (*inode_create) (struct inode *dir,
                     struct dentry *dentry, int mode);
```

- inode security data management

```
int (*inode_alloc_security) (struct inode *inode);
void (*inode_free_security) (struct inode *inode);
void (*inode_post_create) (struct inode *dir,
                           struct dentry *dentry, int mode);
```

Enforcer hooks :

```
[...]
void (*bprm_compute_creds) (struct linux_binprm * bprm);
int (*bprm_set_security) (struct linux_binprm * bprm);
int (*bprm_check_security) (struct linux_binprm * bprm);
int (*bprm_secureexec) (struct linux_binprm * bprm);
int (*sb_kern_mount) (struct super_block *sb);
int (*sb_statfs) (struct super_block * sb);
int (*sb_mount) (char *dev_name, struct nameidata * nd,
                 char *type, unsigned long flags, void *data);
int (*sb_check_sb) (struct vfsmount * mnt, struct nameidata * nd);
int (*sb_umount) (struct vfsmount * mnt, int flags);
void (*sb_umount_close) (struct vfsmount * mnt);
void (*sb_umount_busy) (struct vfsmount * mnt);
void (*sb_post_remount) (struct vfsmount * mnt,
                         unsigned long flags, void *data);
[...]
```

Dummy decider function

```
static int dummy_inode_mkdir (struct inode *inode,
                              struct dentry *dentry,
                              int mask)
{
        return 0;
}
```

Designing a LSM module

- Each module prepares its own mapping framework hook $\longleftrightarrow$ function

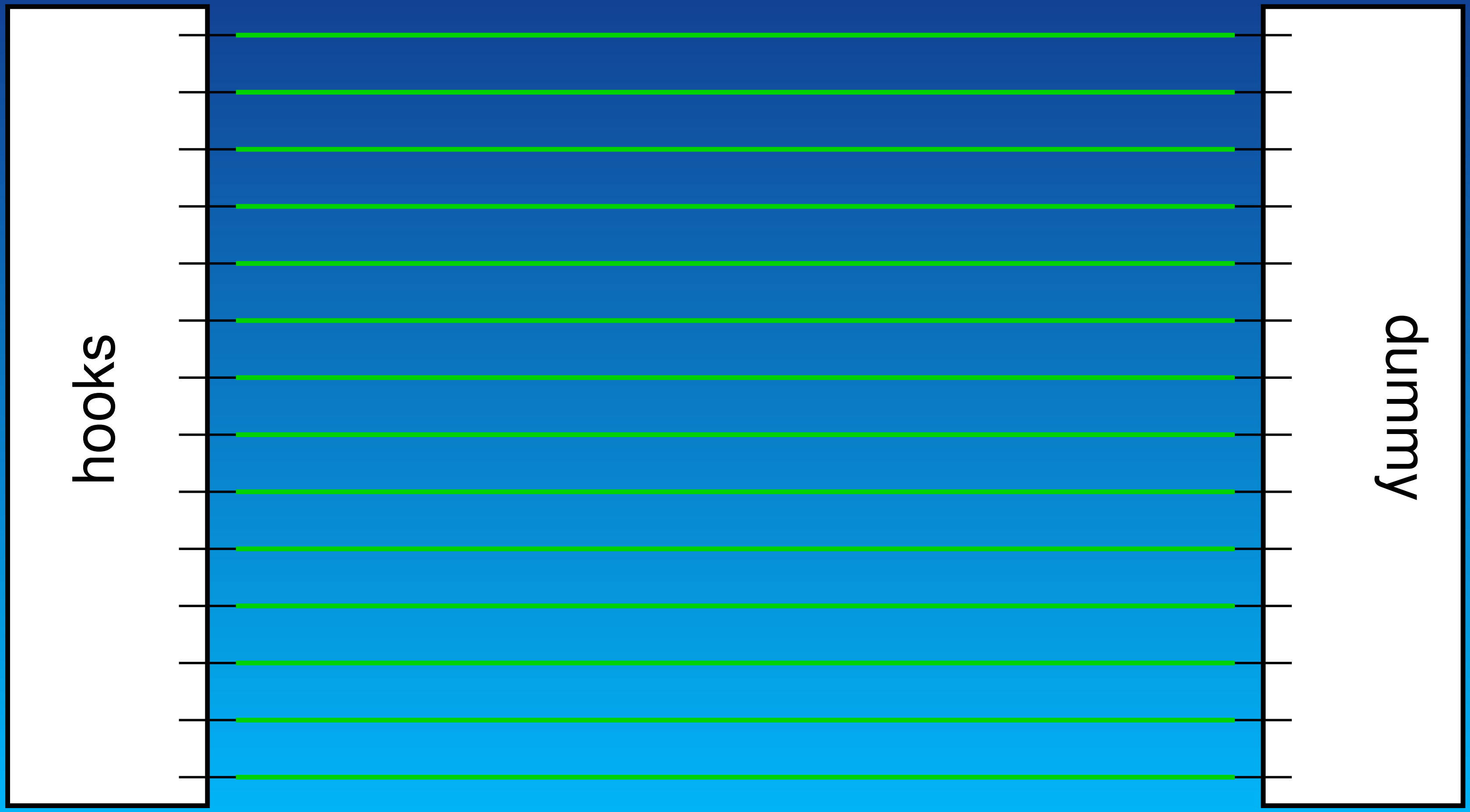
```
static struct security_operations capability_ops = {
        .ptrace =               cap_ptrace,
        .capget =               cap_capget,
        .capset_check =         cap_capset_check,
[...]
```

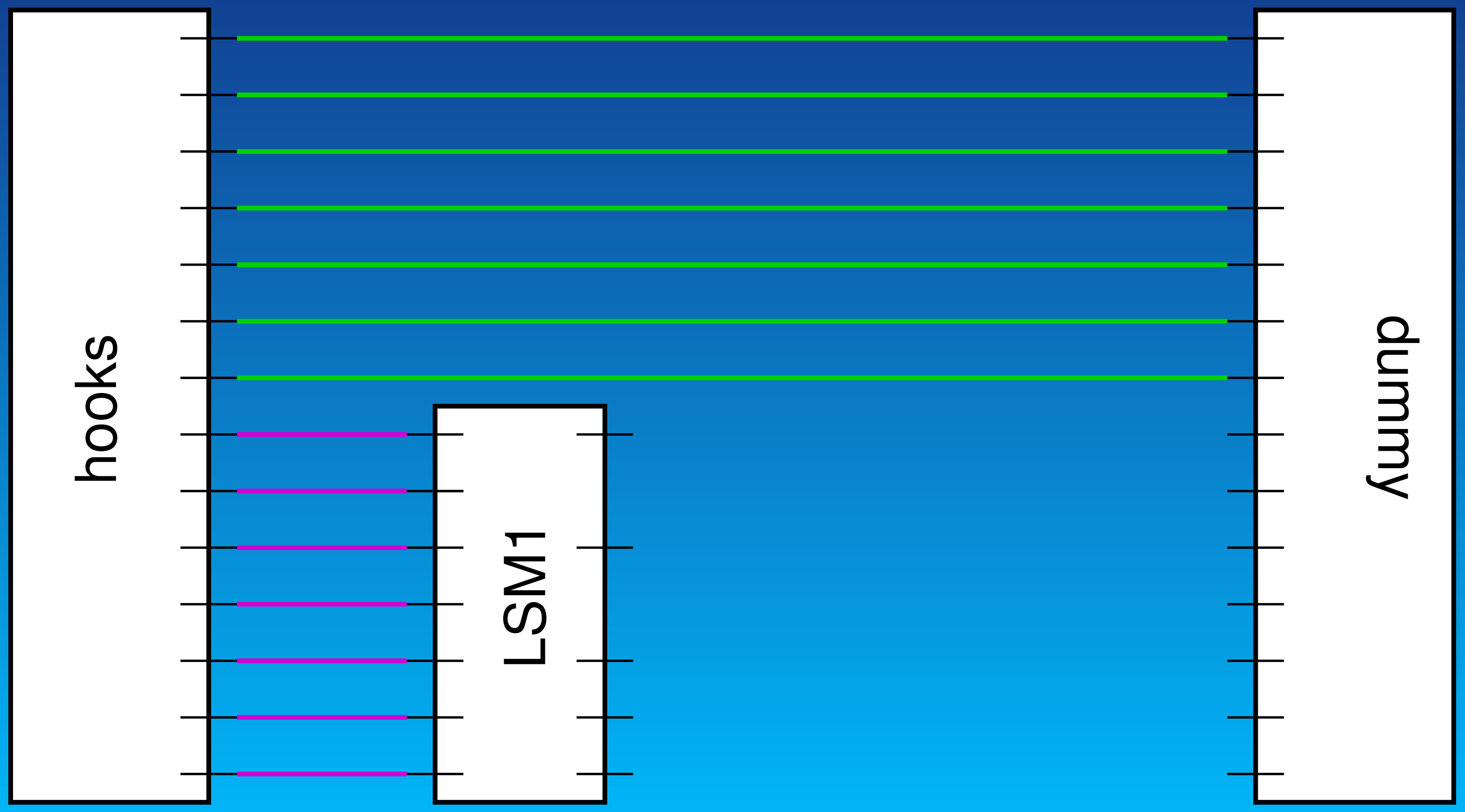
Staking modules

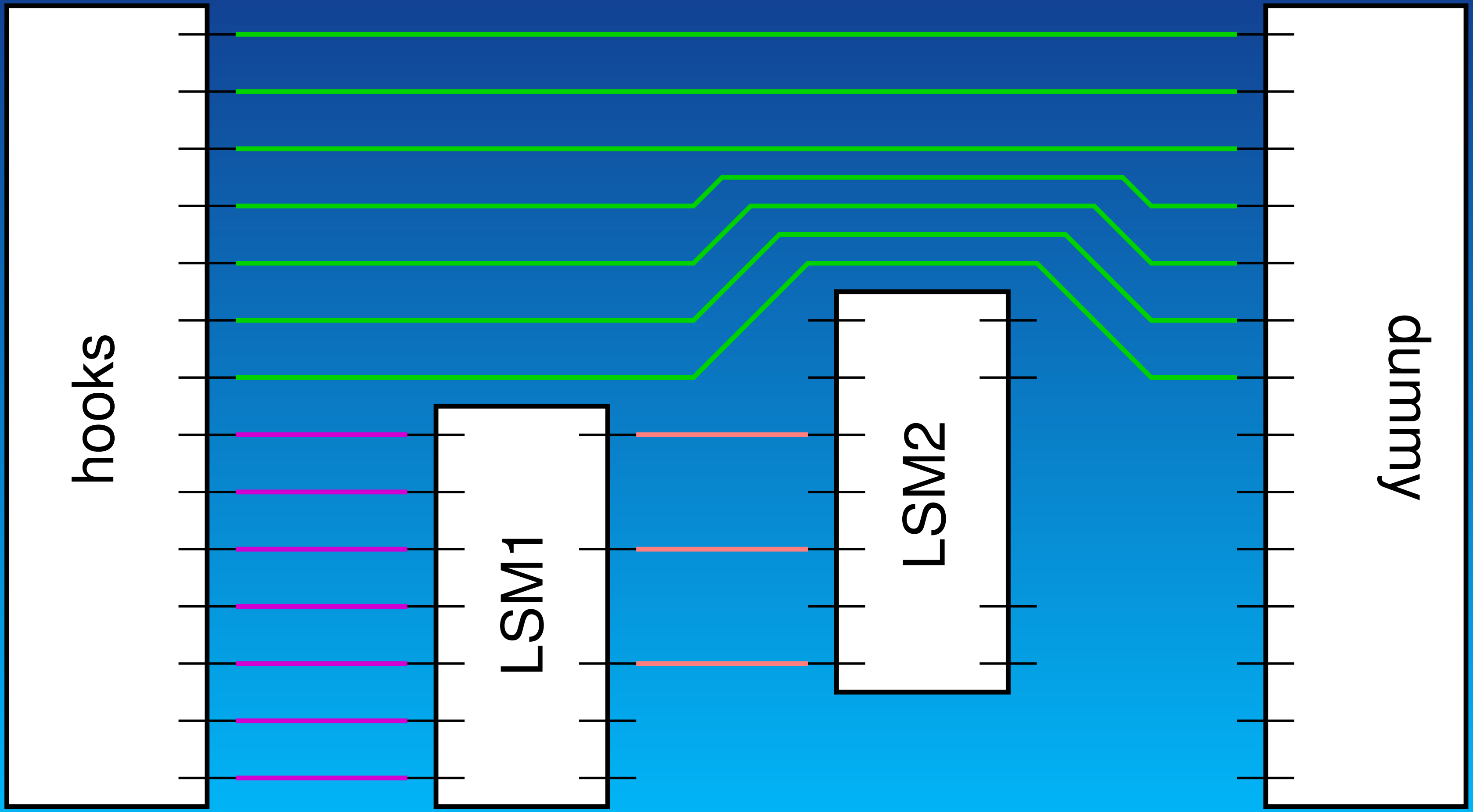
- Possibility to stack two or more security modules
  - module 0 registers to the LSM framework
  - module $n+1$ must register to module n
  - module n is free to ask module $n+1$

Registering a module

```
static int __init capability_init (void)
{
        /* register ourselves with the security framework */
        if (register_security (&capability_ops)) {
                printk (KERN_INFO
                        "Failure registering capabilities with the kernel");
                /* try registering with primary module */
                if (mod_reg_security (MY_NAME, &capability_ops)) {
                        printk (KERN_INFO "Failure registering capabilities "
                                "with primary security module.");
                        return -EINVAL;
                }
                secondary = 1;
        }
        printk (KERN_INFO "Capability LSM initialized");
        return 0;
}
```


hooks
dummy


hooks
LSM1
dummy


hooks
LSM1
LSM2
dummy

Framework consistency

- Nearly impossible to have a theoretical proof
- CQUAL Statistical method

  X. Zhang, A. Edwards et T. Jaeger.

  Using CQUAL for statistic analysis of authorization hook placement

  In USENIX Security Symposium, San Francisco, CA, August 2002.

  ➥ good confidence has been reached

LSM problems

- Stackability issues
  - security data fields not “stack-aware”
  - principal/secondary registation design is heavy
  - ability to support a stacked module is heavy
- POSIX capabilities does not have stacking support
- Only resctrictive access control hooks

LSM module example : root plug (by Greg Kroah-Hartman)

- The decider

```
static int rootplug_bprm_check_security (struct linux_binprm *bprm)
{
        struct usb_device *dev;

        root_dbg("file %s, e_uid = %d, e_gid = %d",
                 bprm->filename, bprm->e_uid, bprm->e_gid);

        if (bprm->e_gid == 0){
                dev = usb_find_device(vendor_id, product_id);
                if (!dev){
                        root_dbg("e_gid = 0, and device not found, "
                                 "task not allowed to run...");
                        return -EPERM;
                }
                usb_put_dev(dev);
        }

        return 0;
}
```

LSM module example : root plug (by Greg Kroah-Hartman)

- The mapping

```
static struct security_operations rootplug_security_ops = {
        /* Use the capability functions for some of the hooks */
        .ptrace =                       cap_ptrace,
        .capget =                       cap_capget,
        .capset_check =                 cap_capset_check,
        .capset_set =                   cap_capset_set,
        .capable =                      cap_capable,

        .bprm_compute_creds =           cap_bprm_compute_creds,
        .bprm_set_security =            cap_bprm_set_security,

        .task_post_setuid =             cap_task_post_setuid,
        .task_reparent_to_init =        cap_task_reparent_to_init,

        .bprm_check_security =          rootplug_bprm_check_security,
};
```

LSM module example : root plug (by Greg Kroah-Hartman)

- The init function

```
static int __init rootplug_init (void)
{
        /* register ourselves with the security framework */
        if (register_security (&rootplug_security_ops)) {
                printk (KERN_INFO
                        "Failure registering Root Plug module with the kernel");
                /* try registering with primary module */
                if (mod_reg_security (MY_NAME, &rootplug_security_ops)) {
                        printk (KERN_INFO "Failure registering Root Plug "
                                " module with primary security module.");
                        return -EINVAL;
                }
                secondary = 1;
        }
        printk (KERN_INFO "Root Plug module initialized, "
                "vendor_id = %4.4x, product id = %4.4x", vendor_id, product_id);
        return 0;

}
```

LSM module example : root plug (by Greg Kroah-Hartman)

- The result

```
$ sudo ls
sudo: unable to exec /bin/ls: Operation not permitted
```

Whatever happens, the future of operating system security seems to rely a lot on kernel level security.

That's all folks. Thanks for your attention.

You can reach me at `<phil@secdev.org>`

These slides are available at `http://www.secdev.org/`