# Kernel Level Security

Philippe Biondi
<phil@secdev.org>
<philippe.biondi@arche.fr>

26th September 2003

**Abstract**

Security is a problem of trust. Having a system that offers services to Internet and that can be trusted is very hard to achieve. Classical security models focus on the physical limit of the machine. We will see that it can be interesting to move the trust limit between user space and kernel space and that it is still possible to enforce a security policy from this trusted place.

We will see some practical ways to have that work done in a modern monolithic kernel (Linux), with some small code examples.

We will also see some other practical aspects with a review of some implementations that exist for Linux kernels, with a focus on the Linux Security Modules (LSM) framework.

## Contents

# 1 Why ?

## 1.1 Context

The IT industry faces lots of threats. There is no need to explicit the motivations to be protected from attacks, may they be so benign as web pages graffitis or more harmful like data steals, resources steals, vandalism, denial of service, tampering operations, etc.

The three fundamental concepts that use to describe the directions we must look at to manage security are *confidentiality*, *integrity* and *availability*.

To enforce these three concepts, we define a set of rules describing the way we handle, protect and distribute information. This is called a security policy.

The security policy is not a technical point of view, but organizational rules that need technical mechanisms to be enforced. We can for example use:

- Tripwire, AIDE, bsign, debsums, . . . for integrity checks
- SSH, SSL, TLS, IPSec, GnuPG, . . . for confidentiality
- Passwords, RSA keys, secure badges, biometric access controls, . . . for authentication

Now the problem is to be confident about each of these technical mechanisms working as they should. Can we be confident that our Tripwire or SSH is not trojaned ? Can we trust our GnuPG ? And, if they work as they are intended to (i.e., if they are not trojaned), how much does they depend on their environment to fulfill their security task ? What if the kernel does not read the key ring GnuPG asked for but one provided by an intruder ?

Security is a matter of trust. As we have just seen, trusting a brick is not sufficient. We must also have an at least equivalent trust in the underlying bricks. If this is not true, we will soon end with a castle built upon sand.

The other problem is that there are a lot of bricks in a castle. Trusting a brick cost a lot, in men, time and money. If there are too many bricks, the construction may become so complex that no human being can understand it entirely, so complex that it will be very probable that human errors or design errors happen.

## 1.2 A new security model

### 1.2.1 Interlude : the mice and the cookies

Let's consider we have some cookies in a house. Let's also imagine that our house, a very old mansion, also hosts mice. We would like to keep our cookies until tomorrow's breakfast, so that we have to prevent mice from eating them.

What can we do for that ?

**Solution 1** We opt to protect the kitchen. The cookies are in the kitchen, so, if we prevent mice from penetrating into the kitchen, our cookies are safe. This is theoretically perfect, but :

- there are too many variables to cope with (lots of windows, holes in the walls, ...)
- we cannot know about all the holes to lock them (especially the one behind the dish machine)
- we cannot be sure there were not any mice in the kitchen before we closed the holes

**Solution 2** We choose to put the cookies in a metal box. This solution, while been theoretically perfect too, has the following practical advantages :

- we, human beings, can grasp the entire problem
- we can "audit" the box

Speaking about the cookies security, we can far more trust the second solution. So, if only cookies are important to us (we will leave the mansion after the breakfast), this should be the retained solution, both in terms of effectiveness and in terms of costs.

We will in fact sacrifice the kitchen for the cookies sake. This seems to be painful (especially for our mate that cooks a lot). But reducing the perimeter of the problem help us approach the perfect intrusion prevention[1] technique. Complexity leads to insecurity and must be avoided.

### 1.2.2 Security models comparisons

We will now focus on the security of a machine.

The usual security model is to consider that the limit between friends and enemies is the physical limit of the box. Anything running on the box is trusted, and everything is build with this assumption (see fig. 1). So, the last limit against intruders, the one that make us surrender if it is broken, is the physical limit of the machine.

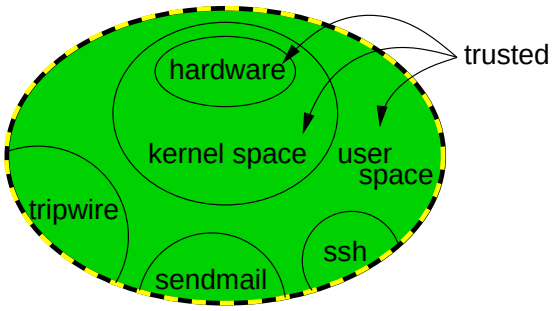


Figure 1: Usual security model of trust

But this limit is very large. There are a lot of applications to take in account, lots of lines of code, lot of entry points, really too much things for what we, human beings, can handle.

Moreover, the fact that everything has to trust everything in the box (with relative separation, though) goes against the compartmentalization principle. This security principle says that things that can run independently should be protected one from the other, just in case one falls to the control of the enemy.

Let's for example imagine that someone cracks into sendmail (see fig. 2). As the perimeter is very large, this is as probable as the penetration of the kitchen

[1] Intrusion prevention, i.e. the fact of doing things so that intrusion cannot happen, is an anti-intrusion technique [HB95], considered as the most effective one, but also the most unreachable one.

by a mouse. In this model, the barrier is at the physical limit of the box. So she is in. She now 0wns a process in which we trust. Nothing is done, now, to prevent her to attack other processes, data, kernel, etc.

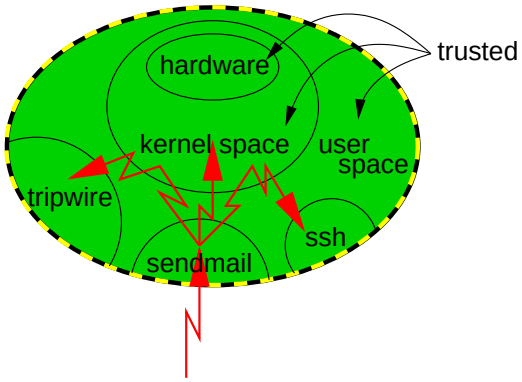


Figure 2: Break-in with usual security model

Now, if we reduce the perimeter to protect, for it to become the kernel space/user space separation (see fig. 3), it looks more like the metal box. There are very few entry points from user space to kernel space. There are far less lines of code that run into our trusted world.

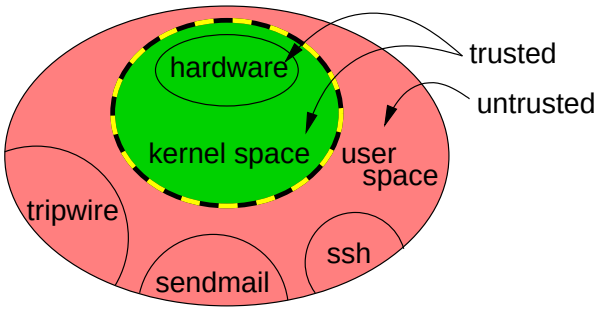


Figure 3: Kernel security model of trust

If someone breaks into the machine, it is bad, but not as bad as previously, because the physical limit of the machine is not our last defense line anymore. We still can defend ourselves (see fig. 4).

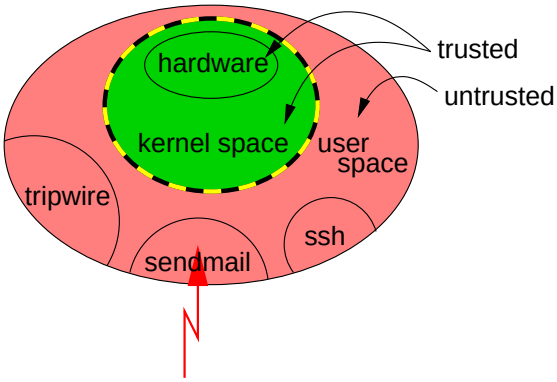


Figure 4: Break-in with kernel security model

Moreover, from where we are (kernel space) we can enforce compartmentalization.

## 1.3 Conclusion

We have just seen that a new way of protecting a machine can be achieved by reducing the last line of defense to the limit between kernel space and user space, instead of the usual physical limit. This *does not* mean that the physical limit has not to be defended anymore. This means that the physical limit must not be the last one. We have to care about protecting the kernel, because we will invest him with a new role : be our last defense to protect the machine and enforce compartmentalization between processes and data.

So, to use this model, we have to modify the kernel in order for it to able to protect itself from outside (user space and everything outside of the physical limit of the machine). This has to be done to the point that we become confident that the kernel will do what we want, and not what an attacker would ask it to do. In particular, this means that, once the kernel has received its orders, it must not listen to anyone, even root, and carry on its mission. For the kernel, root must not be trusted anymore. The orders come from another entity, that can authenticate herself directly to the kernel, without the kernel relying on something else than itself to perform the authentication.

The new mission that is given to the kernel by this entity is to protect other programs and data related to or involved in the security policy. The kernel must be modified to be able to fulfill this new role.

# 2 How ?

We will begin by identifying all the available targets on a machine an all the possible ways to compromise them. Then we will follow with what has to be done to protect the kernel. We will end by the ways the kernel can enforce compartmentalization.

## 2.1 Taxonomy of action paths

The aim of this part is to identify all the possibles paths that lead to a compromise of something. We can model the different components as shown on the figure 5.

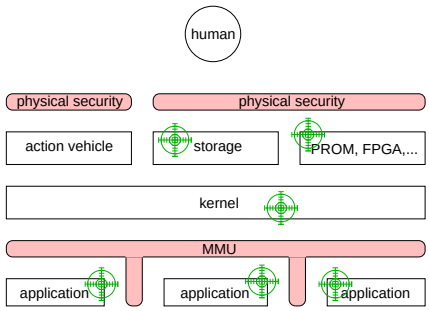


Figure 5: All possible targets

We have at the top, an human being, that will ignite the attack ; machines do not fight by themselves yet. Targets are represented with a green cross : storage devices, like hard disks or flash memories, can be attacked to steal information or resources. PROM or FPGA devices can be trojaned. Applications in memory can have informations like passwords to leak. Kernel can also be trojaned. So, we have all the targets, and some motivations to attack them.

The *action vehicle* component is an interface between real world and logical world, for example a keyboard or a network interface card.

The rounded boxes are for security barriers. Some are physic, like the shielded walls that prevent you from stealing the hard disk, other are logic, like the boundaries enforced by the memory management unit (MMU).

In the following, we will identify thirteen actions paths that can be used to attack a target.

### 2.1.1 Targeting storage or PROM directly

There exists a way to attack storage and PROM devices directly. It means that we have physical access to the box. We use a screwdriver and can extract the disk or the chip to do what we have to do. These are action paths 1 and 2. They have to go through a physical security layer.

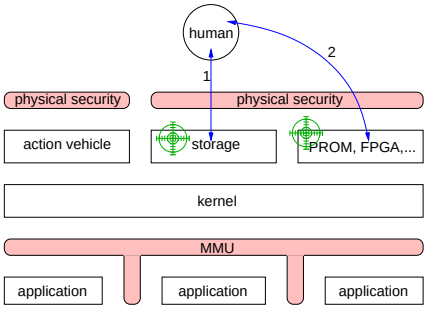


Figure 6: Targeting storage or PROM directly

**Example:** The cracker breaks a lock to reach the box, unscrew it, steal an hard disk and steal all the data present on it.

### 2.1.2 Targeting an application directly

To reach an application, one need to use an action vehicle, which can be a keyboard or a network interface card. This is the path 3, which has to go through a physical security layer. The action vehicle will forward the action to the kernel (4), which in turn, may redirect it (5) to its final destination : an application.

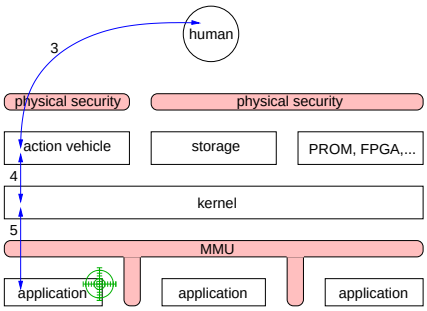


Figure 7: Targeting an application directly

**Example:** The cracker breaks a lock to reach a keyboard. Now that she is on console, every key she types is forwarded by the kernel to the application who owns the current tty.

**Example:** The cracker dials to a modem that will make her be in direct relation with the kernel. The kernel will transmit its stimuli to the application listening to the serial tty. If it is not protected, it can already leak lot of informations, or can be reconfigured or infected.

**Example:** The cracker connects to TCP port through Internet. The kernel get her packets, and allow them to reach the application listening to the TCP port. The application is vulnerable to a buffer overflow, and she injects a shellcode to modify slightly the daemon behaviour.

### 2.1.3 Targeting storage or PROM through an application

If the application has no value, or cannot be exploited, it is still possible to have it do things for us. Through action path (3,4,5), we can give our orders to the application that will then access hard disk through kernel (6,7) or PROM through kernel (8,9). Direct accesses to hardware, like kernel does, are not possible because of the CPU running in kernel mode. Every direct access must be authorized by the kernel, which is the only one that can modify CPU access lists (IDT, GDT and LDT on Intel architectures).

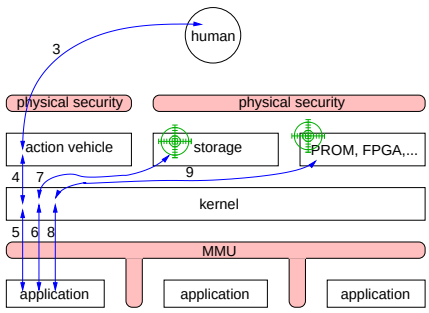


Figure 8: Targeting storage or PROM through an application, ...

**Example:** The cracker had access to a keyboard on a console with a shell opened. She now can access to files on the hard disk, with the consent of the kernel.

### 2.1.4 Targeting an application through an accessible application

If an application that can be accessed is not interesting, it may be able to give access to other applications, either directly through shared memory (10) or using special system calls like `kill()` or `ptrace()` (11,12). If no memory is already shared, which is almost always the case, the MMU will prevent any application to access memory space of another application directly. The attacking application must ask the kernel.

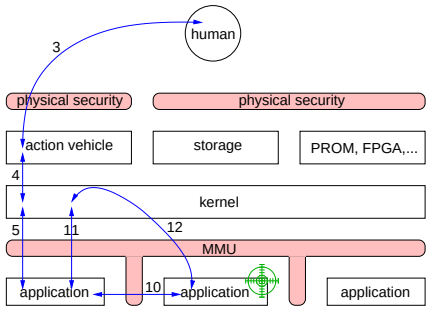


Figure 9: Targeting an application through another application

**Example:** The cracker got access to a shell but wanted to trojan a ssh daemon to get passwords. She uses `ptrace()` to inject a code into `sshd` to modify his behaviour, so that it leaks every password supplied to it.

### 2.1.5 Targeting the kernel

If we want to reach the kernel, either we can reach it through the action vehicle (3,4), or we have to bounce with an application.

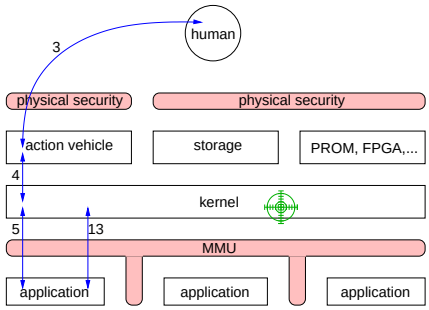


Figure 10: Targeting the kernel

**Example:** Some special key combinations (magic sysreq keys) can leak process tables, registers, etc. to the screen.

**Example:** An error in the network card driver (Etherleak) or in the IP stack (ICMPLeak) make it leak memory. The cracker only need to send packets and see answers, without any interaction with an application, only the kernel. The action path is (3,4).

**Example:** An attacker get a root shell and can attack the kernel through loadable kernel modules or `/dev/kmem`. The path is (3,4,5,13).

#### 2.1.6 Synthesis

All the paths we have previously seen describe the ways an attacker have to take to reach a given target on a machine. Except the physical only attacks that use paths 1 and 2, and the shared memory case that uses path 10, they all go through the kernel. This is a good point in favor of an approach where kernel enforces the security policy.

But not every path can be well filtered. The (4,5) path cannot really do anything, because it is all about data that are interpreted by an application, that the kernel cannot understand.

Moreover, the kernel is directly exposed to attacks. The MMU will protect it against direct access to its memory or to hardware, but it is in direct relation with the action vehicle and can also be attacked using communications ways between applications and itself. We have to make the hypothesis that these interfaces with untrusted world are bug-less. It will never be the case, but we can consider that in a first approximation.

### 2.2 Defending kernel space

#### 2.2.1 Attacks coming through the action vehicle

These attacks are those which hit the kernel from the hardware side : network attacks that target bugs in the network stack, console attacks with magic keys. Kernel cannot do a lot to prevent them from happening, except to be as bug free as possible.

#### 2.2.2 Attacks coming from user space

These attacks are those which hit the kernel from the logic side. They essentially come through system calls, or their use on special files or procfs files.

If we assume that there is no way to exploit the system call interface, the entry points to kernel space, which are opened by the kernel itself, are

- `/dev/mem`, `/dev/kmem`,
- `/dev/port`, `ioperm()`, `iopl()`,
- `create_module()`, `init_module()`,
- `reboot()`

For example, the door to `/dev/mem`, `/dev/kmem` and `/dev/port` can be locked in a single point of the Linux kernel :

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

If you always return `-EPERM` or if you make `capable()` return *false*, these entry points will be closed.

The same can be done for the module insertion control :

```
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;
[...]
}
```

The `reboot()` system call is a special case. It can be used to replace the kernel with a new one, through a complete reboot of the machine. Thus, it is a threat to kernel space. But the rebooting process is an almost user space only process. If nothing is done to prevent this to happen, the `reboot()` system call will be called (and denied) when there is no more processes running.

## 2.3 Filtering in kernel space

### 2.3.1 What to protect

We have to protect a lot of different things, that we can categorize.

**What lives in memory** Lot of very interesting things can be found in memory and nowhere else. For example, the cryptographic key of a crypted partition, which is itself protected by a passphrase on disk, is in clear text in memory. We also find passwords that are in memory only the time to be checked, clear versions of documents, firewalling rules, network communications, interesting facts on what is going on. This must be protected from a cracker's eye.

Moreover, lots of things must also be protected from her hand. She could modify the behaviour of programs, injecting code, to transform them into password collecting programs or key loggers or spies.

**What lives on disks or tapes** Files must be protected from being read or tampered with, to avoid data stealing, behaviour modifications, or disinformation. As files must be accessed for normal operations, this is achieved by compartmentalization.

Meta-data (filesystems, partition tables) or boot loaders must also be protected.

**Hardware** Lots of devices must be protected from crackers. If they can have a raw access to the disk controller, they will bypass every high level control. They must not be able to reach directly any hardware, as they could use it to steal information (for example, grabbing what is in the video card memory), or damage devices. PROMs like the BIOS chip, FPGAs that we find in some audio or video devices, and nowadays reprogrammable CPU are sensitive targets that must be taken out of the reach of crackers.

### 2.3.2 How to protect

We have seen in section 2.1 that every attack, except the physical ones, has to ask the kernel either to mediate the commands, or to give it the permission to reach directly its target (`ioperm()`, ...).

All these things are done via only one interface : the system call interface. Some system calls are too generic for them to be able to enforce the whole security policy (for example `write()`), so that some of the decision process may be delayed to its extensions in device drivers, or any specific functions it may call. But the main idea is that everything will go through the system call interface, and most of the accesses can be processed there.

So, we have to modify consistently the behaviour of the system calls for them to be able to enforce a complete security policy.

**A modular architecture** A good way to do so is to use a modular architecture to control system calls. Enforcer components would be integrated to the original system calls. Each time a system call is issued, the enforcer component

will ask a decider component whether the system call must be granted or denied. The decider component is the one that know about the security policy and take its decisions accordingly. The enforcer component will then enforce the decision of the decider component.

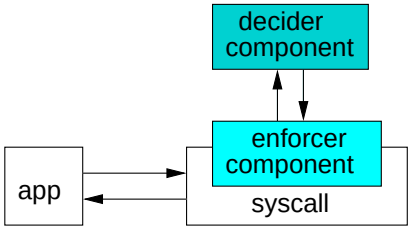


Figure 11: A modular architecture to control system calls

With this architecture, lots of access control policies (DAC, MAC, ACL, RBAC, IBAC, ...) can be implemented, switched and combined, without any change in the enforcer components. Only the decision process is affected.

**The enforcer component** The anatomy of a system call is represented on figure 12. When an application issues a system call, a specific instruction is used (interrupt gate or call gate), so that the processor can safely switch from the user mode to the privileged kernel mode. Then, an entry point which is common to every system calls is executed. Its role is mainly (but not only) to call to the system call that has been asked for. When the system call returns, the dispatcher code gets the hand back, and concludes the call.

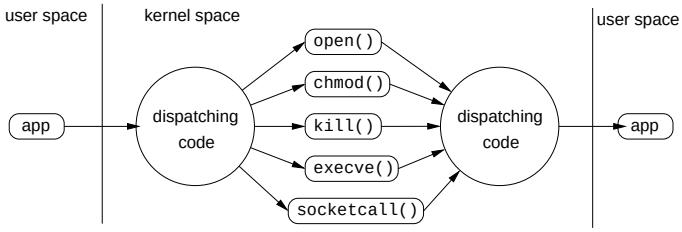


Figure 12: Anatomy of a system call

Knowing that, we can see two ways for adding the enforcer component. Either we do system call interceptions, i.e. we add it to the dispatching code and intercept all the system calls with one modification, or we modify each of the numerous interesting system calls.

**System call interception** The system call interception is done in the dispatcher code. Only one modification has to be done to intercept every system calls. This means a very low cost patch, and a very generic interception mechanism.

The drawbacks are that this piece of code often have to be architecture dependent, and last but not least, that there is a kind of duplication of every system call, because parameters are in their raw form. They have to be interpreted and checked before being submited to the decider component.

Here is, as an example, a part of the enforcer component of the Medusa DS9 project [PZO], from the file `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
```

**System call modification** The system call modification consists in modifying each system call that has to be controled to implement a consistent security policy.

The big advantage of this way of doing is that all system call parameters already interpret and check their parameters. We only have to use them when they are ready and ask our question to the decider component. Moreover, once we have decided to modify a system call, we can do more than only adding 3 lines : we can also tune the system call for a better integration of our access control mechanism.

The drawback is that there are a lot of system calls (more than 200 for Linux) and a lot of them have to be patched.

One example of system call modification in the LIDS [XB] patch shows how much the enforcer component benefits from the checks and decodings done in the begining of the system call. It can for example directly use the `nameidata` structure whereas the parameter was a filename.

```
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) {
```

Here is another example drawn from the Linux Security Modules (LSM) framework [EVW+].

```
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;
```

The set of enforcement components can be seen as a framework where the decider component can be plugged into.

# 3 Implementations

This section will be about some of the implementations that have been done arround these concepts. We will see some of the existing projects in the first section, and we will focus particularly on the Linux Security Modules in the second section.

Note that we will not detail all those projects very much. You will get lot more information on their respective web sites.

## 3.1 Existing projects

### 3.1.1 pH: process Homeostasis

This project is not really related to access control mechanisms because it is an intrusion detection system with couter measure, but it has its place in kernel security. It is based on very theoritical work from Anil Somayaji and Stephanie Forrest [SF00] that has become a real project for the Linux kernel [Som].

Some learning methods are used on the order system calls are issued for given processes. The more a process deviate from its model, the more pH will delay the execution of the system calls. The result is that crackers that try to divert an application from its normal behaviour will have to deal with a slower and slower machine, until been totally blocked.

### 3.1.2 Openwall

The Openwall kernel patch [Des] is a collection of security related features for the Linux Kernel. These features include :

- 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

This collection of patch does not provide any new access control method, as what we have seen previously, but their presence strengthen the operating system with some small kernel behaviour modifications.

### 3.1.3 GrSecurity

GrSecurity [SD] was originally a port for the 2.4 Linux kernel series of the Openwall patch, which worked only for Linux 2.2 kernel series. This patch has evolved a lot. PaX [teaa] has been used instead of the original Openwall non-executable stack protection, bringing with it lot of other neat hardening features like Address Space Layout Randomization (ASLR). An ACL system has been added. Some randomization on PID, TPC XID or TCP sources ports, and auditing code are also present.

### 3.1.4 Medusa DS9

Medusa DS9 [PZO] extends the standard Linux security architecture with an user space authorization server. Its main differences with most other projects are the fact that it uses system call generic interception (see section 2.3.2), and the fact that the decider component run in user space as a daemon.

This latter characteristic make it very versatile regarding the implemented access control mechanisms. Indeed, programming a very wide range of them is easier in user space.

But this design make the decider component less protected by the MMU barrier because it does not lives in kernel space.

### 3.1.5 Systrace

Systrace [Pro] is a very interesting project. It it is avaiable for *BSD kernels and for Linux kernel. It uses system call interception, and is able to control which system calls are permitted, and which parameters can be passed to those system calls. It can also permit privilege elevation on a per system call basis. It is also able to automatically generate a policy for given processes.

### 3.1.6 RSBAC

RSBAC (*Rule Set Based Access Control*) [OFHS] is the meeting of an enforcement framework, named GFAC (*general Framework for Access Control*) and a multitude of access control mechanisms.

The different access control mechanisms are implemented as kernel modules, and two or more can be used at the same time. Among them, we have for example *Mandatory Access Control* (MAC), *Access Control Lists* (ACL), *Role Control* (RC), *Functional Control* (FC), *Malware Scan* (MS), ...

The malware scan access control module will scan every binary when it is executed to check whether it contains any malware. This module is worth being noticed because it is not a very common access control mechanism, and shows how much versatile the access control modules can be.

### 3.1.7 LIDS

LIDS (Linux Intrusion Detection System) [XB] is one of the very early kernel security patches. Some of its specificities are the fact that it has been developped bottom up, i.e. without any theorical model, and also its rough approach regarding operating system hardening.

In particular, it had the approach of placing in the kernel everything it needed to rely on. That is why you can find a little SMTP client and a kind of port scan detector implemented in the kernel. These functionnalities are very controversial, but they can be disabled at compile time. Another controversial functionnality is its ability to make processes invisible.

A sealing mechanism has been set up so that all the privileges needed at boot time (for example, doing a fsck) are definitively removed (as when Egyptian pyramids were closed) at the end of the boot process.

### 3.1.8 LoMaC

LoMaC [Fra] stands for *low water-mark access control*. Its way of working is very interesting and instructive regarding the good old theoritical integrity models à la Biba [Bib77].

LoMaC considers two integrity levels : high and low. For the initialization, some directories are tagged as high integrity. The other directories have a low integrity level. Each time a binary is executed, the process inherits its integrity level from the directory where the binary was located. Whenever a high integrity level process opens a low integrity file or an internet socket, it becomes a low integrity process. A low integrity process cannot open a high integrity file or signals a high integrity process.

With these very simple and obvious rules, that govern the life of the entire system, we can mathematically prove by induction that the integrity is always ensured.

The simplicity of the model is really appealing, but some exceptions soon arise. Indeed, with this model, we cannot, for example, maintain secure logs. Log files must be protected. So they must have a high integrity level. So the syslog program must have a high integrity level. Programs that generate logs can be low integrity processes, in particular those that open sockets. So, the unix socket used to collect logs must be a low integrity (special) file. Thus, if syslog reads this file, it must become a low integrity process, and cannot write into the log files anymore. This problem cannot be resolved with the simple model. Exceptions have to be defined. LoMaC can give a special property to some binaries, which consists in permitting them to read low integrity files while being high integrity processes.

### 3.1.9 SE Linux

Here are very few words about the *Security Enhanced Linux project* (SE Linux) [Teab]. This is a NSA funded project, even if most of the people working on it do not work for the NSA anymore.

The project is based on the *Flexible architecture security kernel* (Flask). This is an access control framework, very similar to RSBAC's *Generic Framework for Access Control* (GFAC) (see section 3.1.6).

The most unique feature of SE Linux in regard with other projects is the attention that has been paid about the change of the access control policy in the middle of operation, and in particular the access revocation.

## 3.2 Linux Security Modules

The Linux Security Modules project was born after a SE Linux presentation by Peter Loscocco at the San Jose Kernel Summit, in 2001.

Linus was convinced that something had to be done regarding access controls, but did not want to choose specifically one project. They are so many access control mechanisms and so many needs that it is not possible to find a *one size fits all* mechanism.

He decided instead to develop a generic framework, modular enough to enable people to write access control mechanisms as loadable kernel modules (LKM). The LSM project was born.

The framework took the shape of a set of hooks in the Linux kernel, in order to branch on them different kinds of access control mechanisms. We now find running implementations of SE Linux, LIDS and DTE that use this framework.

### 3.2.1 Security hooks

As we have seen previously, LSM use the system call modification method. A set of hooks has been inserted at key places in the Linux kernel. Most of them are decision hooks, which means that they are called to take a decision about granting or not the execution of a given operation. But, in order to make a decision, keeping track of some data related to processes, inodes, etc. is sometimes necessary.

Thus, other kind of hooks have been inserted whose goal is to provide entry points at other key locations to allocate and free security data in some well chosen structures like the `task_struct` structure or the `inode` structure.

### 3.2.2 Stacking modules

We have seen previously that RSBAC was able to run multiple access control mechanisms at the same time.

LSM also have a mechanism that enable them to run more than one security module at the same time. This is called stacking. Indeed, once the first module is in place, the second module will have to register itself to the first. The first module may not support to have another module on its back, but if it does so, it will be responsible for transmiting decision questions to it, and will have the possibility not to listen to it.

### 3.2.3 Testing the LSM framework consistency

One of the biggest problems of access control policies or frameworks is to be able to prove their consistency. In the case of very complex systems like Linux and the LSM, the consistency cannot be proven, but some interesting works [ZEJ] using statistical methods have brought a good confidence in the LSM framework.

## Conclusion

The near future of Linux kernel security seems to be with the LSM, which have been integrated in the 2.6 kernels. But the LSM framework does not enable everything to be done. This framework is very oriented on access control mechanisms. They still lack some auditing capabilities. Moreover, other security mechanisms like those present in PaX, will be very difficult to integrate, even if they also have a good impact on security. So these approaches need to be mixed up.

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

## References


[Bib77] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report 3153, MITRE, 1977.

[Des] Solar Designer. Linux kernel patch from the Openwall Project. `http://www.openwall.com/linux/`.

[EVW+] Antony Edwards, Chris Vance, Chris Wright, Greg Kroah-Hartman, Huagang Xie, James Morris, Lachlan McIlroy, Richard Offer, Serge Hallyn, Stephen Smalley, and Wayne Salamon. Linux Security Modules. `http://lsm.immunix.org/`.

[Fra] Timothy Fraser. LOMAC: MAC You Can Live With. `http://opensource.nailabs.com/lomac/`.

[HB95] L. Halme and R. Bauer. Aint misbehaving - a taxonomy of antiintrusion techniques. In *Proceedings of the 18th National Information Systems Security 14 Conference*, pages 163–172. National Institute of Standards and Technology/National Computer Security Center, 1995.

[OFHS] Amon Ott, Simone Fischer-Hübner, and Morton Swimmer. Rule Set Based Access Control. `http://www.rsbac.org`.

[Pro] Niels Provos. Systrace - interactive policy generation for system calls. `http://www.citi.umich.edu/u/provos/systrace/`.

[PZO] Milan Pikula, Marek Zelem, and Martin Ockajak. Medusa DS9 security system. `http://medusa.fornax.sk/`.

[SD] Bradley Spengler and Michael Dalton. GrSecurity. `http://www.grsecurity.org/`.

[SF00] Anil Somayaji and Stephanie Forrest. Automated response using System-Call delays. pages 185–198, 2000.

[Som] Anil Somayaji. pH: process Homeostasis. http://www.scs.carleton.ca/~soma/pH/.

[teaa] PaX team. Homepage of the PaX team. http://pageexec.virtualave.net/.

[Teab] SE Linux Team. Security Enhanced Linux. http://www.nsa.gov/selinux/.

[XB] Huagang Xie and Philippe Biondi. Linux Intrusion Detection System. http://www.lids.org/.

[ZEJ] X. Zhang, A. Edwards, and T. Jaege. Using CQUAL for static analysis of authorization hook placement. In USENIX Security Symposium, San Francisco, CA, August 2002.