Implementing Noninterference in the Linux Kernel

Pablo Mata
Flowgate Security Consulting
Argentina

Formal Methods in Security – ReSeCo Workshop – November, 2007

Flowx Prototype

- Developed at Flowgate Security Consulting
- In this presentation we will discuss all the details about the implementation of a basic prototype for the security model described previously.
- We seek for three primary properties:

SECURITY-EFFICIENCY-COMPATIBILITY

Linux Security Modules

The LSM is a framework that:

- allows the Linux kernel to support a variety of computer security models.
- avoids favoritism toward any single security implementation.
- provides hooks all over the kernel to intervene in critical security situations.
- adds new fields in kernel structures to allocate security information.

We tried to use all the potential of this technology, but some requirements forced us to patch the kernel source tree.

Implementation Strategy

- Noninterference is implemented in a security module.
- The L and H memories of the model is implemented as concurrent processes, called s-siblings.
- Input is shared between s-siblings by buffering it in a buffer per security level.
- Labeling of resources is implemented using extended attributes and the SysFS filesystem.
- Although not included in the formal model, user authentication is implemented through a trusted path.

Extended Attributes (xattr)

- Were introduced with the 2.6 kernel and are supported by ext2, ext3, JFS, ReiserFS and XFS filesystems.
- Allow dynamic new attributes in the inode structure of a given file.
- Particularly, useful to add, persistently, security labels to files.

Security Labels

- The sc structure represents the security level of all entities in the system:

```
struct sc {
    u32 level;
    u32 categories[NUM_CATS];
};
```

- Particularly, the fields ***(void *) security*** and ***(void *) i_security*** point to the security structure we choose for tasks and inodes, respectively.

First step: *read* and *write* syscalls

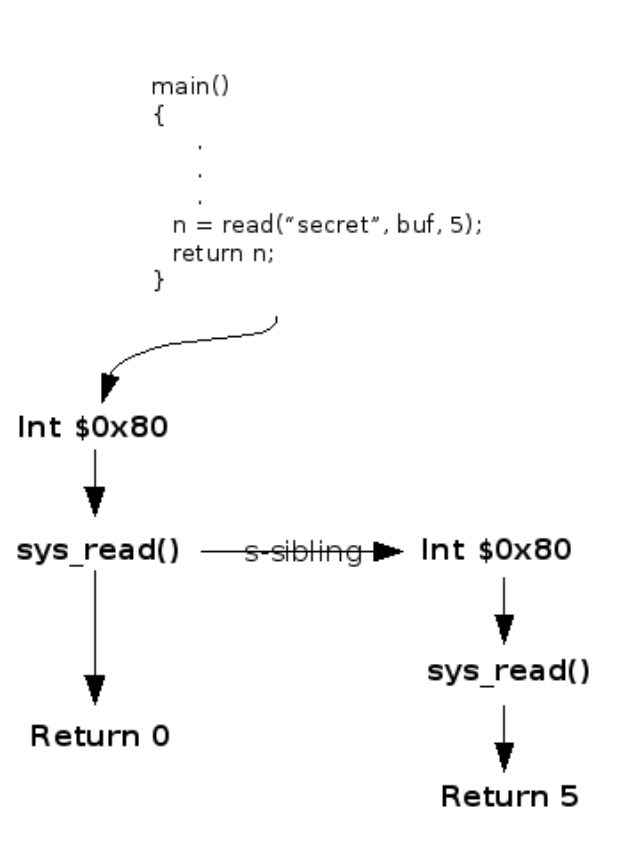
Let P be a process with security class C_P, and A a file with class C_A, then:

- Read:
  - P can read A iff C_P dominates or equals C_A
  - Otherwise, check if P does not have an **s-sibling** with class C_A.
    - If it has, return as if an end-of-file has occurred (return 0).
    - Else create a new s-sibling and return like above.

First step: *read* and *write* syscalls

- An **s-sibling** is a process exactly equal to the process that invoke its creation, differing only in their instruction counter. The s-sibling's program counter will point to the instruction immediately before the one that produce its creation, that is, the read syscall.
- The security level of the new s-sibling will be set as the security level of the file that his brother could not access.

First step: *read* and *write* syscalls



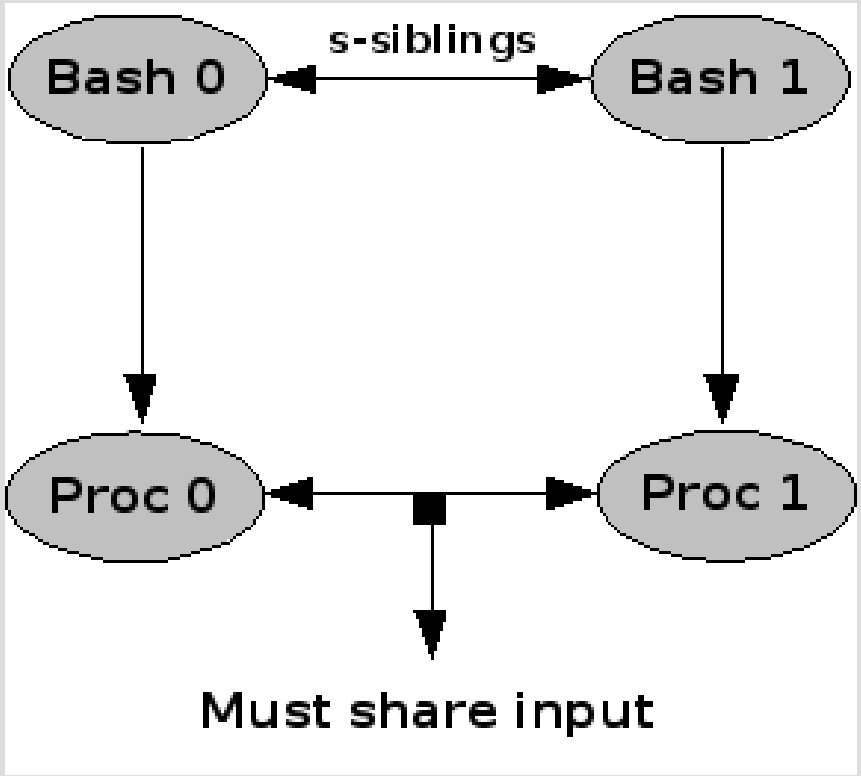
- The s-sibling and his brother take different paths for returning to user space.
- The hook to be used is called file_permission(), included in vfs_read() and vfs_write() helpers.

First step: *read* and *write* syscalls

- Write:
  - A file can only be written by the s-sibling which access class is equal to C_A. If such s-sibling does not exist, then it can write only the one, whose security class is:

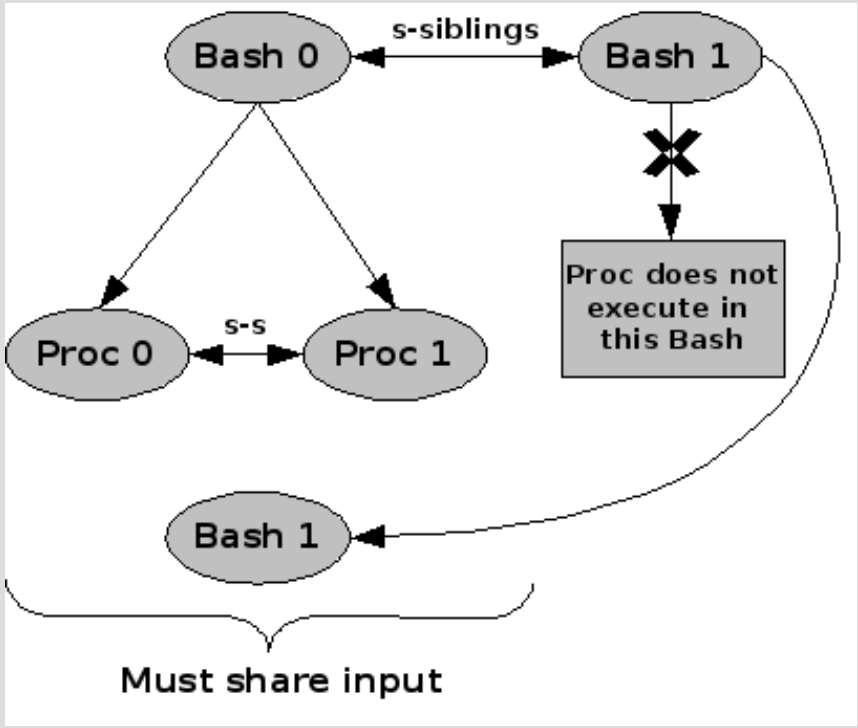
    sup {sc: access class | sc is the access class of an s-sibling that is dominated by C_A}

Shared input



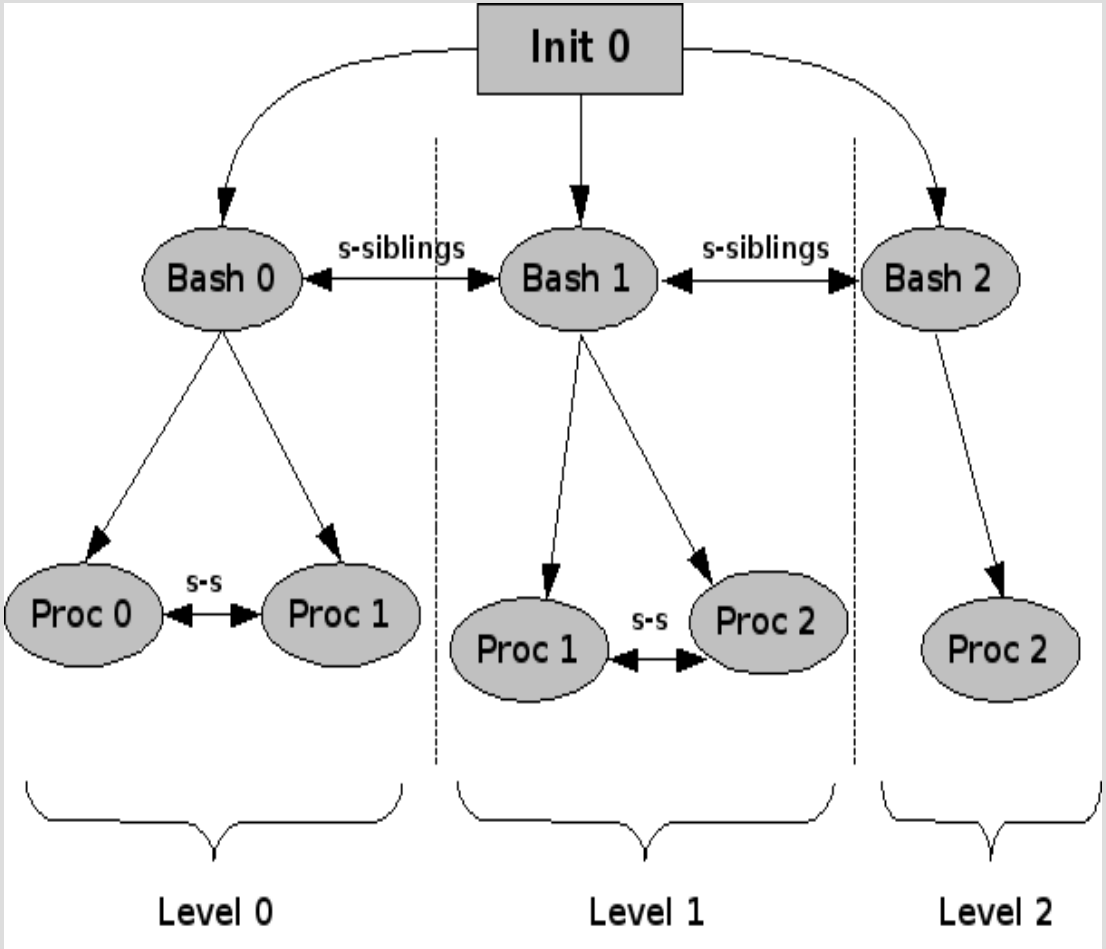
- Low level users must not be aware of s-siblings existence.
- s-siblings have to share low level input to follow the same execution path.
- Although, this is not always true, it is valid in the majority of cases.

Shared input



- Sometimes, the processes sharing input are different, but we do not care.
- Not a security breach.

Shared input

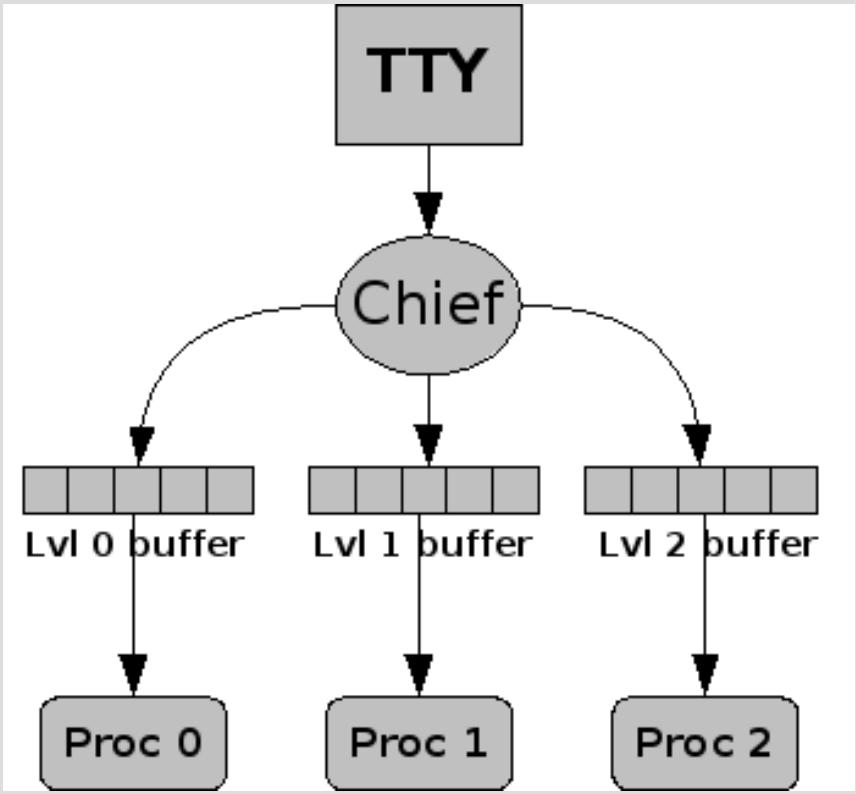


Input must be shared between different levels in the same session.

Shared input

- Each region in the tree will have one or more foreground processes and only one of them will have true access to the tty device.
- This “chief” process will store all received data in a buffer for each level in the session.

Shared input




```
vfs_read(...)
{
  .
  ret = security_file_permission(...);
  .
  .
  .
  if (!ret) {
    n = file->f_ops->read(...);
  .
  .
}
```

Decides who is the chief

tty->read() if chief.

flowx_buffer_read() otherwise.

Shared input

- The chief will be the foreground process in the group whose level is the same as the tty output.
- The chief is the only one that can invoke ioctl request on the output device.
- We must apply the same logic in the select() syscall, for compatibility sake.

Other syscalls

- Now we have to intervene another system calls to ensure data secrecy.
- The idea is to divide directories and to include secret files in secret folders.
- We must ensure that unprivileged processes does not know about secret folders' content.

Other syscalls

- sys_mkdir() → security_inode_mkdir()
- sys_rmdir() → security_inode_rmdir()
- sys_open() → security_inode_permission()
- sys_creat() → security_inode_creat()
- sys_mknod() → security_inode_mknod()
- sys_rename() → security_inode_rename()
- sys_unlink() → security_inode_unlik()
- sys_symlink() → security_inode_symlink()
- sys_stat() → security_inode_permission()

Other syscalls

Let P be a process with security class C_P, and A a file whose directory is D with level C_D, then:

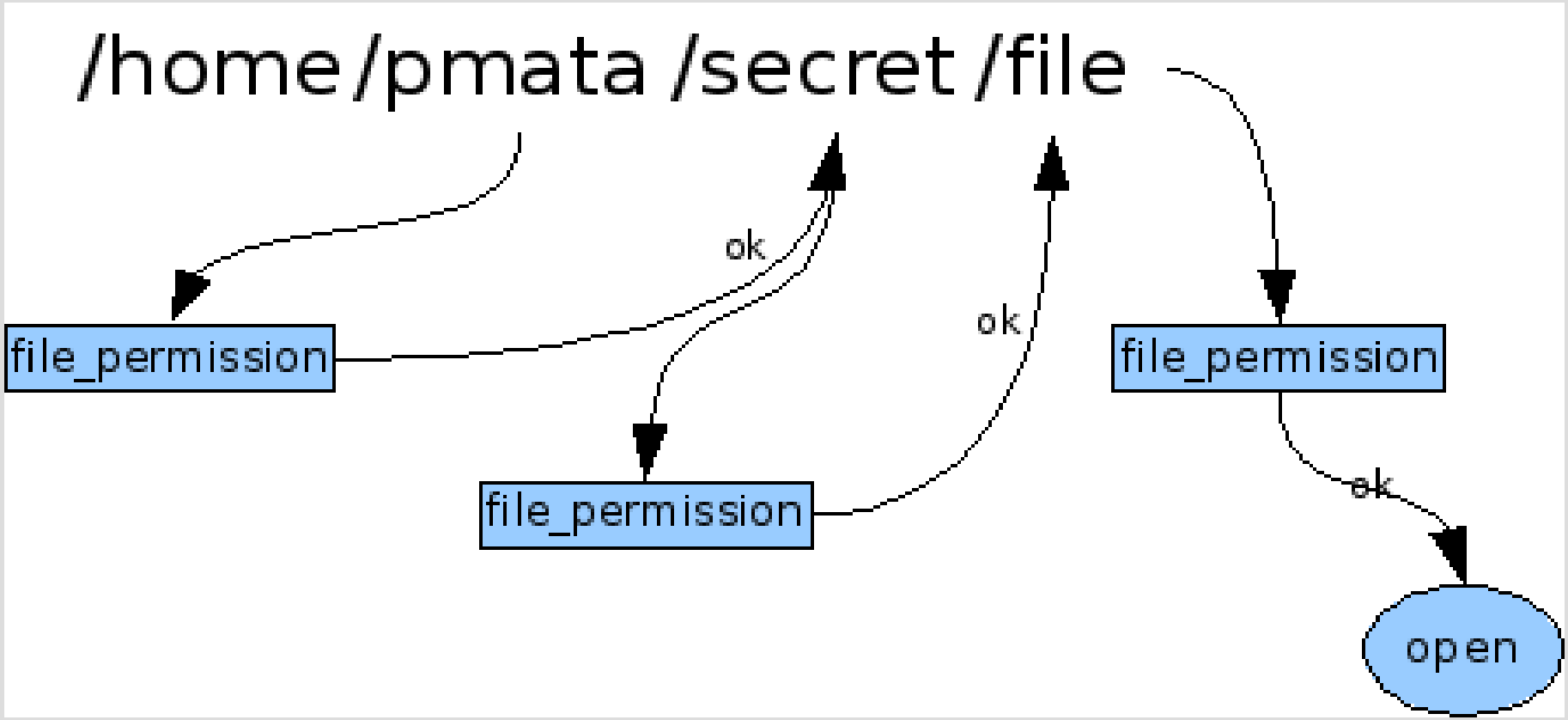
- Open(A):
  - P can open A iff C_P dominates or equals C_A
  - Otherwise, check if P does not have an **s-sibling** with class C_D.
    - If it has, return as the file was opened, returning a file descriptor pointing to /dev/null.
    - Else create a new s-sibling and return like above.
- Stat(A): like open but returning 0.

Other syscalls

- Create/mkdir/rmdir/rename/unlink(A):
  - If C_P equals C_D → ACCESS GRANTED.
  - Otherwise, check if P does not have an **s-sibling** with class $C_{D.}$
    - If it has, return as the operations went successful, (usually 0).
    - Else create a new s-sibling and return like above.

Other syscalls

Pathname lookup



Labeling Devices

- We must set labels to device nodes representing their access levels.
- Two different sc structures are needed, one for output and one of input.
- Problems emerge because, generally, device nodes are create in /dev which is a temporal filesystem.
- We need a configuration file to make device's levels persistent.
- If the device is not in the file, the minimum access class is assumed.

Labeling Devices

- Problems:
  - Files cannot be read or written from the kernel. For this, we have to take a process context or create a new one. However, this is bad seen in the community.
  - We do not want to add more syscalls.
  - We need a mechanism to change security labels dinamically in execution time.
  - We must not add new ioctls request.
- To summarize, we need a way to interchange, dually, information between the kernel and user space.

Labeling Devices: Device Driver Model

- The Device Driver Model is a new feature in 2.6 kernels, which provides a mechanism to represent devices and to describe its topology in the system.
- Design, originally, to implement an intelligently, power management.
- It relies on special kernel structures to represent this topology efficiently.

Labeling Devices: kobjects

- These structures are:
  - **kobject**: is the heart of the DDM. It is similar to a class in object oriented programming. They are not useful in their own, so they are embedded in larger structures.
  - **ktype**: is associated with each kobject and controls what happen when the kobject is no longer referenced and its representation in sysfs.
  - **kset**: represents a group of kobjects all embedded in structures of the same type. It contains sets of kobjects.
  - **subsystem**: is a collection of ksets that constitute a sub-part of the system.

Labeling Devices: Sysfs

- Sysfs is a virtual temporal filesystem that allows users to see the entire topology of all devices in the system, as a simple filesystem.
- The magic behind sysfs is to link kobjects with directories through special field in all the kobject structure.
- Because kobjects infrastructure already formed like a hierarchic tree: the DDM, and with kobjects linked to directories entries of the kernel, the implementation of sysfs was trivial.
- Through its ktype structure, every kobject can export different attributes to the sysfs filesystem, providing methods to their access.
- By these methods, a driver developer can map kernel structures in a regular file. Modifying one of these files also modifies the structures and vice versa.

Labeling Devices: Our Approach

- We create a new subsystem in sysfs.
- During boot time, when the kernel detects a new device it sends an event through a socket to notify this situation. A user mode program (UDEV) listening to this socket is in charge of creating the device node.
- Once the new device node is created, we use a hook in mknod system call, which will run a user mode program to register these device in our sysfs subsystem.
- The program will read the configuration file and will set the corresponding access levels.
- If no entry in the file is found, the device is considered non trusted and its labels are set to have the minimum security level.
- Once initialized, only the security administrator can change a device access class.

Secure Login

- Every user needs to have a security access class and this will be represented by the level of the terminal he will be given after the login process.
- We have to assign new levels to ttys depending on who has logged in. This is a very delicate issue concerning security in our system, so we need:
  - A new login program, initially running of behalf on the security administrator and the changing its UID to the corresponding user.
  - Reflect in the kernel that the security administrator is more privileged than root in security terms and in setuid syscalls.
  - A way to ensure that always our login is the one executed and not a malicious one. For this, we need a trusted path.

Secure Login: Trusted Path

- We use the sysrq key system, which consists on a combination of specific keys to produce a special respond from the kernel, no matter what the system is doing.
- The kernel offers an interface to add new keys to this system.
- Our trusted path will be invoke by pressing “Alt + Print Scree + A”.
- Basically, this will:
  - Kill all process in the session.
  - Create a new kernel thread from scratch.
  - Assign to this thread the current tty and make him the session leader.
  - The thread will execute our secure login program to authentify the user and set the tty output and input levels according to that user.

Secure Login: Trusted Windows

- A special mechanism for the kernel and programs from the TCB to securely communicate messages to the user.
- Implemented in the last line of the current terminal.
- Modifications in the TTY and VGA drivers.
- New syscall that programs from the TCB can use to print in the trusted windows.

Conclusions

- **Efficiency**: the performance penalty when the module is active are insignificant with modern computer hardware.
- **Compatibility & usability**: almost all basic Linux utilities and user applications show to adapt perfectly to our security policies and kernel modifications. Others, like some text editors need a slight variation in the way they are invoked.

Future Work

- Future work will be focus on:
  - providing security control in the kernel IPC (interprocess communication) mechanism.
  - controlling network traffic through sockets.
  - adapting the prototype to be used with the X window server.