# Pastures: Towards Usable Security Policy Engineering

Sergey Bratus, Alex Ferguson, Doug McIlroy, Sean Smith

Institute for Security Technology Studies
Department of Computer Science
Dartmouth College

# A practitioner's look at the field

Powerful formalisms exist:

- Older: "The Orange Book", Bell–LaPadula, Biba, ... (MLS)
- Newer: FLASK, type enforcement (SELinux)

But:

- Admins, app writers (even vendors) appear to shun them.
- Security-conscious practitioners create and use other solutions.

We try to:

1. Review existing design space from the usability viewpoint.
2. Formalize ideas already in use.
3. Combine them in a new approach to a set of more usable policy primitives.

# At a first glance

Available high assurance systems have large and complex policies.

- Hard to write and maintain
- Crafted by trial and error

## Example

SELinux in Fedora Core 3:

- Policy makes use of M4 macros
  - 2000+ lines in core M4 macro base
- Mediates 160+ *domains* ($\sim$ daemons and applications), 145 *operations* ($\sim$ syscalls)
- 227275 lines in the *strict* policy

# A 10,000 miles review of SELinux internals (1)

- Each process and each resource (file) operated on has a *domain* or *type* label.
  - `task_struct`, `inode`, `super_block` structs have a `(void *) security` member that points to respective SELinux policy structures.
- System call hooks (LSM) check the process and file labels for each mediated operation, and deny access unless permitted by the policy:
  - `allow sshd_t sshd_key_t:file {getattr read};`
  - `allow ssh_keygen_t urandom_device_t:chr_file {getattr read};`
- All objects in the root filesystem are pre-labeled
  - `/etc/ssh/ssh_host_key -- system_u:object_r:sshd_key_t`

# A 10,000 miles review of SELinux internals (2)

- New files and processes are assigned labels based on the labels of:
  - for files, parent directory and creating process,
    - `file_type_auto_trans(ssh_keygen_t, etc_t, sshd_key_t, file)`
  - for processes, parent process and executable file,
    - `daemon_base_domain(ssh_keygen)` → `type_transition sysadm_t ssh_keygen_exec_t:process ssh_keygen_t;`
- Permitted operations can be specified in terms of type *attributes* (sets of labels, e.g., `sysadmfile`.)
  - `type sshd_key_t, file_type, sysadmfile;`
  - `allow sysadm_t sysadmfile:file { getattr read write create unlink ... relabelfrom relabelto };`

# The ssh-keygen example

ssh_keygen_t is the type of the ssh-keygen program when run by the admin to generate the host key or at install time.

## file_contexts/program/ssh.fc:

```
/usr/bin/ssh-keygen -- system_u:object_r:ssh_keygen_exec_t
```

## domains/program/ssh.te → policy.conf:

```
type ssh_keygen_exec_t, file_type, sysadmfile, exec_type;

allow initrc_t ssh_keygen_exec_t:file {read { getattr execute }};
allow sysadm_t ssh_keygen_exec_t:file {read { getattr execute }};
allow ssh_keygen_t ssh_keygen_exec_t:file entrypoint;

type_transition initrc_t ssh_keygen_exec_t:process ssh_keygen_t;
type_transition sysadm_t ssh_keygen_exec_t:process ssh_keygen_t;

allow ssh_keygen_t ssh_keygen_exec_t:file {read getattr lock
execute ioctl};

...
```

# Observations

## SELinux case study

One policy language and mechanism for

1. Access control, Integrity, Confidentiality
2. Host intrusion alerts
3. Inevitable admin exceptions & delegation

- Using one language for all goals causes policy bloat: good expressive power for some goals, not enough for others.
- Structure imposed by M4 macros is implicit.
- Flow properties are not “first class” language objects and must be derived (*Apol & other Tresys tools, SLAT, PAL, etc.*)

# Audit2allow

## A “first cut” at a type description:

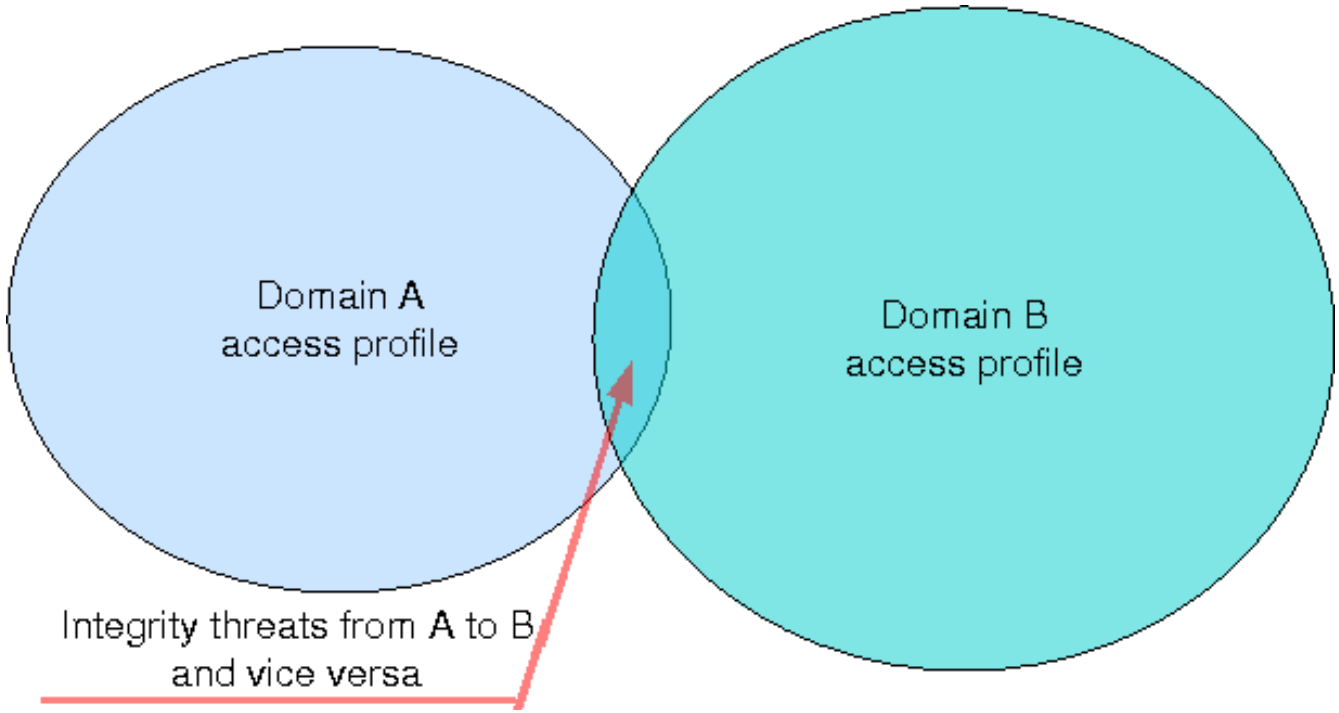
A tool and approach to generating new types:

1. Run application in non-enforcing mode, record accesses that would be denied.
2. Generate policy from the log trail by allowing relevant accesses.
3. Repeat until all legitimate code paths are covered.

## Problems:

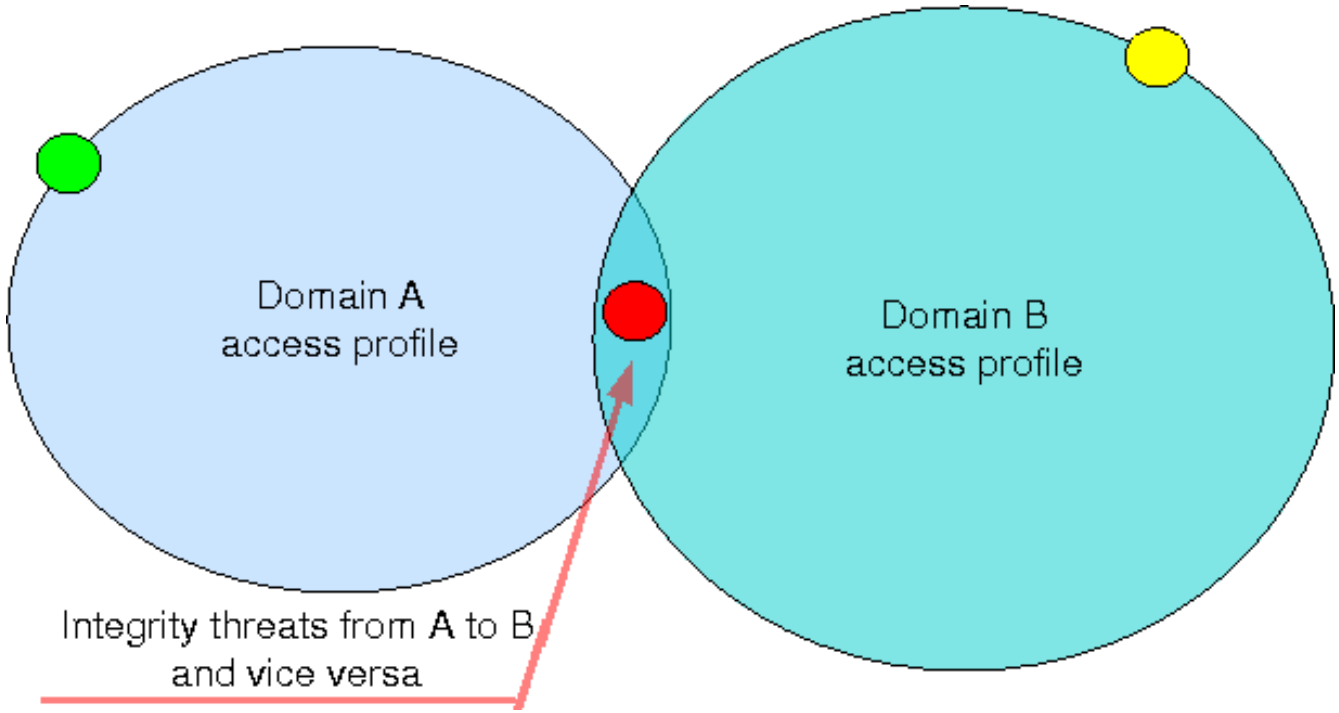
- Friendly network: too little diversity.
- Real network: what is the average time to attack?

# Policy complexity vs. admin concerns



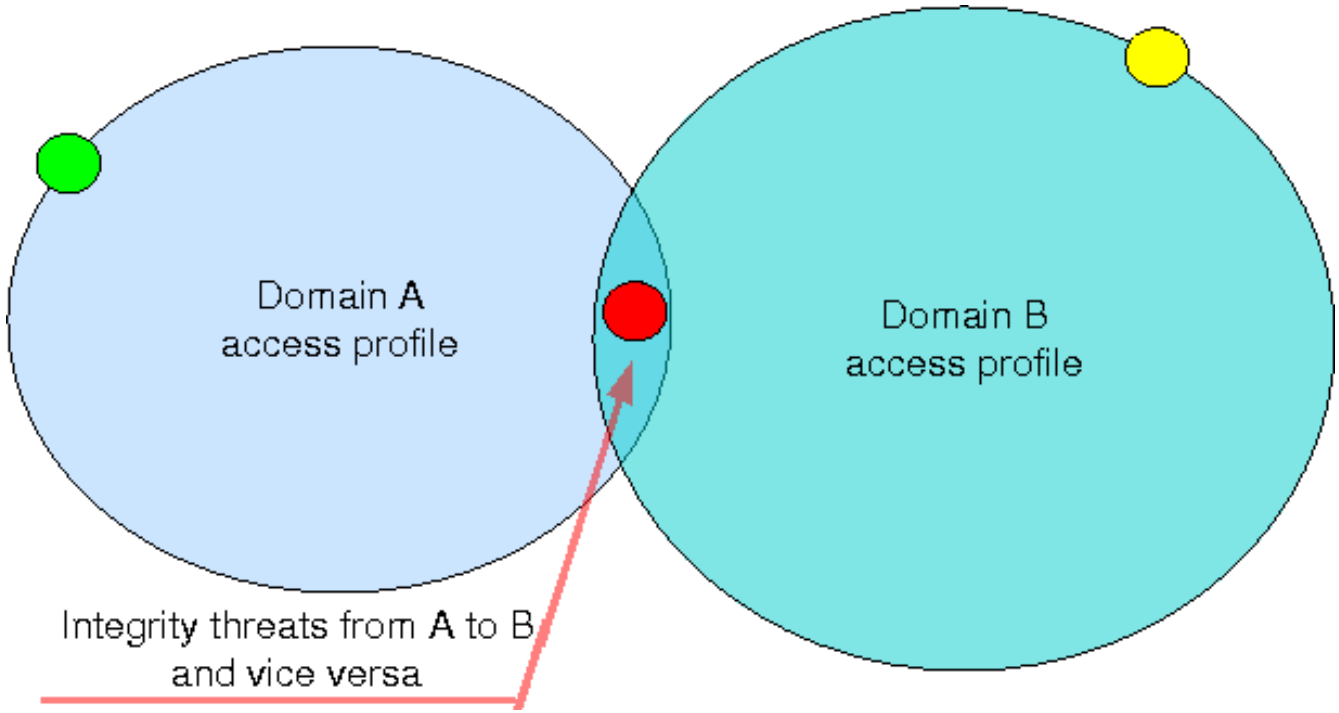

- Not all profile violations are of the same concern;
- ...but all may kill the offending process.
- Again, all actually legitimate accesses will need to be explicitly allowed.

# Policy complexity vs. admin concerns




- Not all profile violations are of the same concern;
- ...but all may kill the offending process.
- Again, all actually legitimate accesses will need to be explicitly allowed.

# Policy complexity vs. admin concerns




- Not all profile violations are of the same concern;
- ...but all may kill the offending process.
- Again, all actually legitimate accesses will need to be explicitly allowed.

# Observations summarized

## Principal usability obstacles:

- Any degree of integrity protection requires a large and complex policy that *profiles all allowed accesses*.
- Policy mimics software's execution profile, and becomes as complex as software itself (but without software engineering tools).
- No protection before profiles are compiled.

The reason for these inconveniences is fundamental:

## The "Least Privilege" principle

Deny all accesses and operations unless explicitly allowed by the policy.

... but is it really a *security goal* in and of itself?

# Questions of goals and trade-offs

In an actual operational environment:

1. Can the cost of developing an access profile for a new application or daemon that needs be installed be somehow postponed without compromising *integrity* of the rest of the system?
2. While a policy addition is still being “debugged”, how bad are crashes due to a legitimate code path not covered by the policy?
3. VMs are conceptually simple, but where is the trade-off between that simplicity and maintaining multiple OS instances?

# Alternatives?

## Popular practical solutions:

1. BSD jails (*"chroot on steroids"* + private IP address)
2. Linux Vserver (separate non-communicating process contexts, separate virtual filesystems)
3. Solaris 10 Zones (no sharing by default, high privilege granularity, *resource access can be inherited*)
4. OS emulation approaches: User Mode Linux (UML), Xen, etc.

## The lesson: **Usability** $\sim$ **Virtualization**

- Simpler management, *less attention to partition insides*.
- Flows between partitions are absent *by default*.
- Integrity is provided by *separation* rather than by profiling.

# Vserver review (1)

## At a glance

- One kernel: no virtual machines, no OS emulation.
- $\sim$ BSD Jails, $\sim$ Solaris 10 Zones.
- Security through by-default isolation.

## Processes

- Processes are assigned to Security Contexts and labeled with a *context ID (XID)* label (added to `task_struct` and elsewhere).
- *These is no communication between processes in different contexts* – separation enforced by system call hooks.
- The system starts in *Host context* (XID $= 1$).
- Each context has its own root filesystem (via *chroot* $+$ extra anti-escaping measures. But: see “Unification”.

# Vserver review (2)

## File systems

- Each context has its own FS root.
- Inodes store the context ID (XID) of the creating context
- Inode access checked against XID (except for the special *Spectator* context).
- No pre-labeling: untagged files are OK to access, modified files get the creator's XID.

But:

Maintaining separate filesystems is expensive:

- Library and utility upgrades,
- Security updates,
- Matching configurations between partitions.

# Vserver Unification

## "A truly great idea"

- Share files between contexts when they are unlikely to change (libraries, standard binaries)
  - Reduces admin effort
  - Reduces inode caches and memory mappings.
- The contexts' filesystems start out populated with special type of hard links: *immutable but unlink-able*.
- Once changed from within and context, the link is *removed and replaced* with a new file, private to a context.

Combines the benefits of:

- a single FS/namespace for admin tasks,
- keeps file changes private to a context.

# Observations

We see a pattern in the use of virtualization-based solutions:

1. Isolation and separation by default
2. The de-facto access policy exists as *exceptions* to (1):
   - Some files can be shared between Vserver contexts
   - Some host filesystems can be mounted “live” from inside a VM (UML, BSD jails)
3. A unified admin view of partitions’ filesystems saves effort.

- Not all would-be policy violations are of the same concern w.r.t. the actual security goals
- It is often preferable to let a process continue after an access violation rather than kill it outright. (*Liang, Sekar*, ACSAC ’03, “Alcatraz”)

# Our approach

## Copy-on-write (COW) as a fundamental policy primitive.

Protected processes run in isolated "pastures".

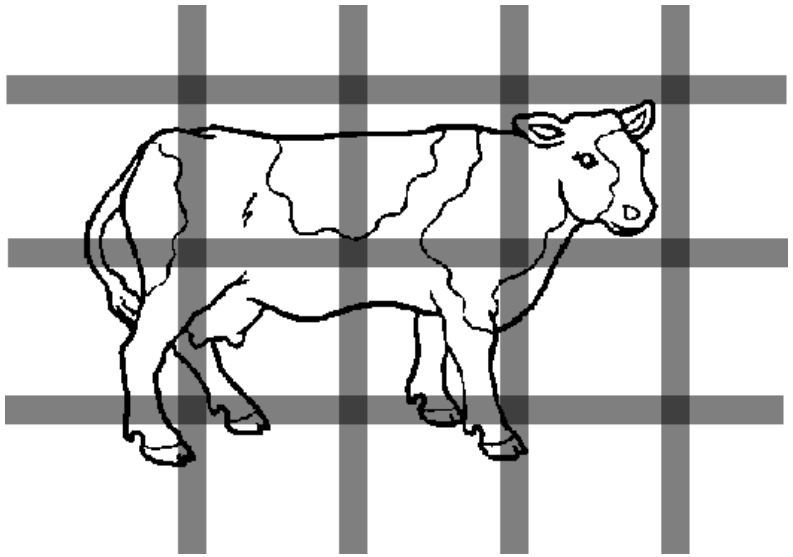
1. All accesses not explicitly allowed or denied by a policy statement result in COW duplication of the accessed object, rather than a denial.
2. Sharing objects between pastures becomes a basic policy statement type.

## Advantages:

- New progams can be introduced without *audit2allow* risks.
- Write flow properties are specified rather than derived.
- Less crashes: EACCESS errors fatal under SELinux but not critical to security goals are no longer fatal.

# What's in a name?

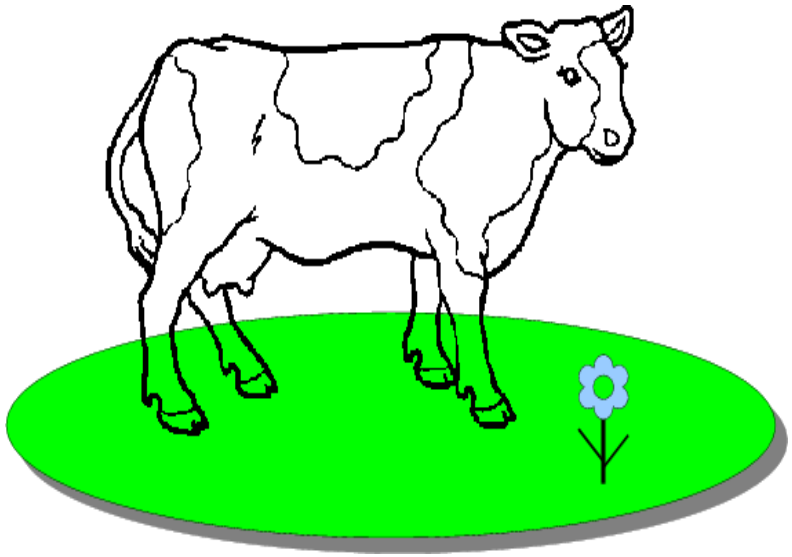
We thought of calling our compartments "COW jails"...



But that sounds cruel...

# What's in a name?

We thought of calling our compartments "COW jails"...

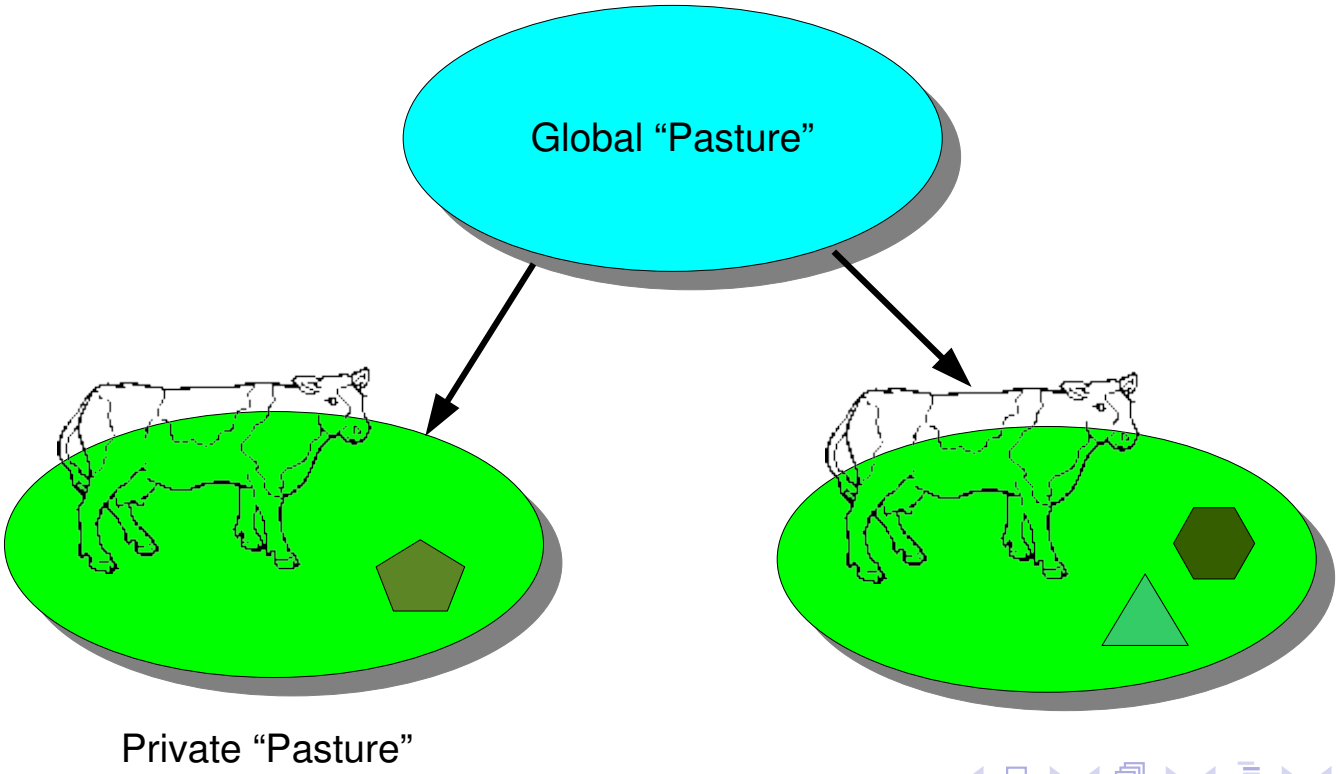


So we called them "pastures" instead ;-)

# “COW pastures” (1)

The policy specifies that certain processes are to be placed in a “pasture”.

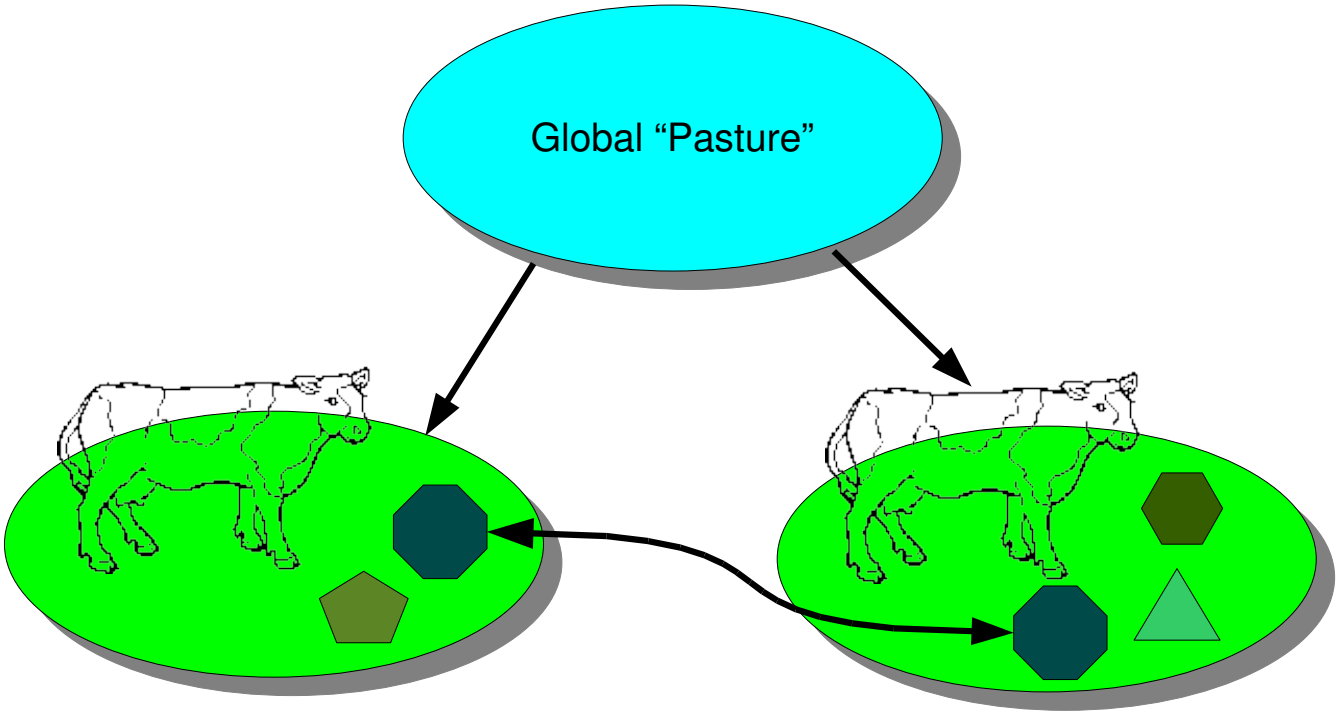
For a process placed in a pasture, any file modifications are private by default. Any *write*-type access results in creating a private copy:



# “COW pastures” (2)

If processes in different “pastures” need to share resources, the policy *explicitly* specifies the allowed sharing.

References from sharing pastures will resolve to *the same underlying object*. This object may be global or private to those pastures:



# Policy elements

## Unidirectional write-flow

$G \rightarrow A$: changes made to the namespace of pasture $G$ by processes running in $G$ are visible to processes in the pasture $A$, but not vice versa.

## Example

Introducing a new server/application into a set of trusted ones.

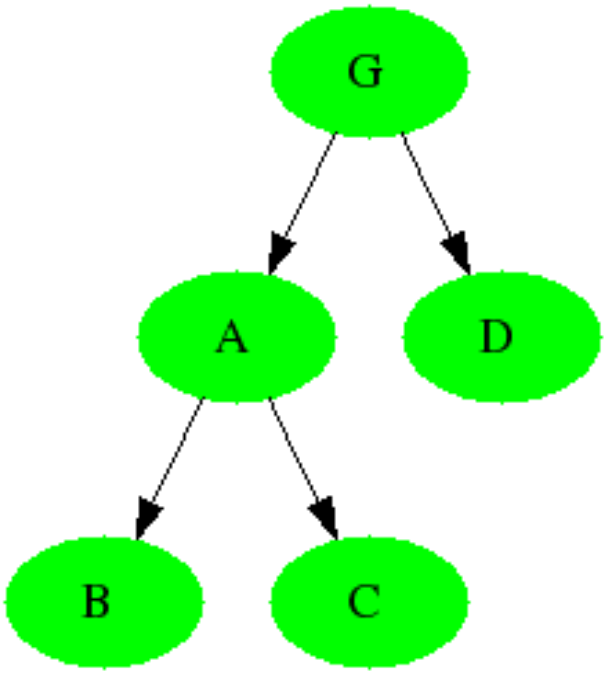
## Bidirectional write-flow

$A \leftrightarrow B$ on *file_spec*: specified files are shared between pastures $A$ and $B$.

## Example

Servers sharing access to a database or filesystem.

# Examples




## Placing processes into pastures:

```
/usr/sbin/httpd => A
/usr/sbin/ftpd => A

/opt/matlab => D
/opt/lic-server => D
```

## Sharing files:

```
A <-> B:
/path/to/file1
/path/to/file2
```

# Prototype implementation

- A patch for the Linux 2.6.12 kernel.
- Modifications to the *ext2* filesystem.
- Uses */proc* pseudo-filesystem for control:
  - */proc/pastures/* – active pastures
  - */proc/<PID>/pasture* – pasture ID
  - */proc/pastures/<pasture_id>/cows/* – links to COW-ed files
- *Write*-flows only.

## Future work

- Creating parametrized pastures “on the fly” (by user ID, *sudoers*-style aliasing)
- Read flow control policy primitives.
- Support for other filesystems?
- LSM integration?

# References


LIDS: The linux intrusion detection system, http://www.lids.org/.

Linux vserver project, http://linux-vserver.org/paper.

L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker.
A domain and type enforcement UNIX prototype.
*Computing Systems*, 9(1):47–83, 1996.

K.-H. Baek and S. W. Smith.
Preventing Theft of Quality of Service on Open Platforms.
In *IEEE/CREATE-NET SecQoS 2005*, September 2005.

W. Boebert.
The lock demonstration.
In *Proceedings of the 11th National Computer Security Conference*, 1988.

J. Dike.
User-mode linux.

P.-H. Kamp and R. N. M. Watson.
Jails: Confining the omnipotent root.

P.-H. Kamp and R. N. M. Watson.
Building systems to be shared securely.
*ACM Queue*, 2(5), July/August 2004.

D. Langille.
Virtualization with freebsd jails, 2006.

Z. Liang, V. Venkatakrishnan, and R. Sekar.
Isolated program execution: An application transparent approach for executing untrusted programs.
In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2003.

P. Loscocco and S. Smalley.
Integrating flexible support for security policies into the linux operating system.
In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*. The USENIX Association, 2001.

D. Price and A. Tucker.

# Projects that provided the details

- SELinux and the NSA Sample policies + FC3 policies
- Linux Vserver, *http://www.linux-vserver.org*
- BSD Jails, *http://docs.freebsd.org/44doc/papers/jail/*
- Jeff Dike, *User Mode Linux*, *http://user-mode-linux.sf.net*
- Liang, Sekar, *Alcatraz* (a *ptrace*-based private sandbox for an application).
- Solaris 10 Zones, *http://www.sun.com/bigadmin/content/zones/*
- . . .

# Questions?

Thank you!