Pastures: Towards Usable Security Policy Engineering

Sergey Bratus, Alex Ferguson, Doug McIlroy, Sean Smith
Dartmouth College

Abstract

Whether a particular computing installation meets its security goals depends on whether the administrators can create a policy that expresses these goals—security in practice requires effective policy engineering. *We have found that the reigning SELinux model fares poorly in this regard, partly because typical isolation goals are not directly stated but instead are properties derivable from the type definitions by complicated analysis tools. Instead, we are experimenting with a security-policy approach based on copy-on-write "pastures", in which the sharing of resources between pastures is the fundamental security policy primitive. We argue that it has a number of properties that are better from the usability point of view. We implemented this approach as a patch for the 2.6 Linux kernel.*

1 Introduction

Computing systems typically depend on their operating systems to enforce trustworthy behavior. Architects and administrators describe this behavior in terms of various security goals, such as protecting the integrity and confidentiality of data. To achieve these goals, architects and administrators should be able to configure them with security policies that actually express these goals. Thus we reach a point when policy engineering becomes a critical issue.

The problem of engineering usable OS security policies has been worked on for decades. However, we claim that age does not imply maturity. The continuing trouble with securely configuring real systems for real applications in the real world demonstrates the absence of the accepted and effective best practice that would come with a solved problem. Approaches based on formal models do not appear to have gained much traction with system administrators and defenders, outside of a few small specialized communities. Satisfactory approaches to securing systems in manageable ways are still being pursued and discussed[1], and we will later review several such approaches that enjoy de facto practitioner acceptance (most notably *vserver* and the use of *BSD jails* for building virtual systems). We offer our own work as a contribution to this effort.

In recent years, older ideas have re-emerged, with NSA's *Security-Enhanced Linux (SELinux)* (e.g., [11]) considered by many to be the de-facto best-of-breed for those wanting a high-assurance but contemporary OS. Unfortunately, SELinux has a high cost from the point of view of usability: its monolithic and awkward policy structure makes it difficult for programmers to configure and maintain it for real-world applications – and difficult for stakeholders to trust that the resulting policy actually confines system behavior to "secure" operation only.

Previous studies (e.g., [4]) led us to the conclusion that the principal usability obstacles for the SELinux policy approach are:

- any reasonable degree of integrity protection requires a large and complex policy, essentially profiling all the allowed accesses for protected applications;
- by following the application execution profile, such a policy becomes as complex as the original software – without any corresponding software engineering tools to keep it manageable;
- no protection can be afforded before such profiles are compiled, a labor-intensive procedure.

All of these obstacles are ultimately due to SELinux's reliance on the fundamental and time-honored design principle of *denying access if it is not explicitly permitted by the policy*.

In addition to these obstacles, checking if a SELinux policy satisfies information flow goals is a non-trivial task, because flow properties of the SELinux model cannot be expressed explicitly in its policy language, but are instead derived from the type specifications, i.e. access profiles. Thus even though flow goals are often of primary interest, they are not first-order policy objects that can only be derived from a rather large number of access statements.

We explore an alternative approach, based on two principles:

[1] In particular, practitioner conferences, including the so-called hacker conventions like Defcon and Blackhat, often feature talks on exploratory methods for protecting common server software known to be vulnerable.

- All accesses not explicitly allowed or denied by a policy statement result in a *copy-on-write (COW)* duplication of the accessed object, rather than a denial.

- Flow properties are directly specified by the policy rather than derived from other kinds of statements. Specifically, sharing of resources between environments isolated by default becomes a basic policy primitive.

In order to implement these principles, we assign protected applications to *COW pastures* or, for short, *pastures*, where a *pasture* is a security context similar to a *vserver* context or a *BSD jail*, which contains private copies of files modified by processes assigned to it. The term, of course, is a pun on COW, the copy-on-write operation that adds new objects to a pasture. The term also suggests the permissive nature of a pasture as compared to a "jail": the point of pastures is to allow programs to proceed by creating private copies of objects modified by their writes not explicitly listed in an access profile, rather than terminating them for a violation of the profile.

2 Our approach

2.1 Revisiting types

We consider the set of SELinux types as a set of intersecting "access jails", defining their allowed interactions. Indeed, the purpose of SELinux policy analysis tools is mostly to *derive* a description of such interactions (e.g., the information flow between types). It seems that a direct specification of allowed flows combined with the default integrity protection given by the copy-on-write namespace isolation approach, could be much more concise and intelligible, being closer to the actual security goals of many administrators. The principal reduction in complexity will come from the conceptual simplicity of namespace isolation-by-default.

2.2 Motivating Examples

Let us consider two motivating examples.

Server protection

An administrator runs a number of trusted and possibly interacting servers on a system wishes to introduce a new piece of third party software, say a license server. Not being sure of the security properties of this software, the admin would like to take steps to protect the integrity of his other servers from its possible interference, whether unintentional or due to hostile transactions.

Client protection A rich client such as a web browser needs to access and interact with many OS resources, and read and create multiple files, including files in the user's home directory (cache, javascript, cascading stylesheets, media files etc.). A vulnerability in one of its modules could damage the user's private files. Due to its significant footprint, it typically needs a complex security profile with many special cases.

Users nowadays urgently need to maintain separate roles (and effective namespaces) for "browsing" and "working", or face dangers to more valuable private files from the much less valuable "web" files.

Internet Explorer's reasonable idea of "zones" probably owed something to such considerations, but was ruined by the "integration" with the OS and other design decisions that the current "phishing" epidemic has taken such broad advantage of.

Hence client software is best run in an isolated compartment. However, for most user activities, full isolation is not an option (e.g., a web browser is needed to interpret the very working files that we would like to protect from being damaged by it, should it misbehave). However, we note that the number of files that actually need to be imported back to the user's working set is typically much smaller than the total number of files read or written within a session.

Adjusting the policy to fit the quickly changing working environment is hardly a usable solution when the policy is to deny by default all operations that are not explicitly allowed. A denial could well lead to an error and loss of work in the current session.

Thus using a "deny by default" policy means too many *administrative interruptions*, when the user has to switch roles to perform an administrative action, and then spend some time restoring his working context. To avoid such interruptions users often work with elevated privileges all the time. An extreme example was older MS Windows where administrative actions could not be easily taken without logging out and back in as the Administrator.

2.3 The proposed solution

The essence of our approach is to allow groups of related programs to run in "pastures" with the same namespaces as the original system, by default creating private copies of resources when they are modified and describing the cases when this should *not* happen in the policy that enumerates allowed sharing and communication between pastures. In each pasture, the bindings in the namespace thus point either to the original object (if the policy specifies this), or to the private object (created by default if written to by the pasture).

Pastures sacrifice some of SELinux's power for detecting misbehavior but avoid the burden of detailing "good" behavior before protection becomes effective. The effects of bad behavior are managed by keeping them private to the program's pasture.

Advantages of pastures are

- new progams can be introduced in protected manner, eliminating the need for the risky *audit2allow* profiling step;
- architects and administrators can *directly specify* information flow properties;
- the overall complexity of the policy is thereby reduced; and
- access error conditions fatal under SELinux enforcement but not critical to security goals are no longer fatal.

3 The implementation

3.1 Overview

Processes start in pasture environments specified by a policy mapping of the pair *(command, user)* $\rightarrow$ *pasture ID*. The *command* is either an absolute path to an executable or a *sudoers(5)*-style alias to a command with options[2]. Our code resolves the mapping of a process to its appropriate pasture at the point of the *exec* system calls.

The system creates new pastures as needed. Each new pasture starts out with the namespace identical to that of the base system, perhaps with some part of the namespace explicitly excluded by the policy.

The implementation affects name-resolution mechanisms, such as *namei* for the filesystem. Name-resolution routines now take one extra argument, the ID of the pasture, and return the most specific binding that exists for this pasture. If a file or directory has been modified within a pasture privately, the namespace mappings of the particular pasture and the other pastures diverge at the corresponding directory level, and *namei* henceforth produces the private copy of that file, as the most specific for that pasture.

Besides the filesystem, we need consider other namespaces as well, such as those of network ports and various system and network stack constants. Our current implementation is limited to filesystems, but the same approach should be generalized to these other namespaces. To confront the practical challenges connected with virtualizing network stacks, we plan to draw on the wisdom of the *Vimage* project[3] for FreeBSD [15].

[2]The sudoer-style matching and aliasing are not yet implemented

[3]http://www.tel.fer.hr/zec/BSD/vimage/

3.2 Pastures policy primitives

3.2.1 The one-directional arrow

Suppose G and A are two pastures. The policy statement $G \rightarrow A$ means that *changes made to the namespace of pasture G by processes running in G are visible to processes in the pasture A, but such changes by A are not visible in G.* (If the policy also specifies the converse $A \rightarrow G$, we write $G \leftrightarrow A$; we consider this simpler case later.)

Let us return to our previous example of a sysadmin introducing a third-party application A to a system that runs a number of trusted servers, which he wants to protect from possible interference by A. In other words, he wants his trusted processes to affect A as they normally would, but he does not want A to affect them back in any ways other than the few prescribed. Eventually, as the admin convinces himself that A is well-behaved, he will bring A into the trusted fold, but before that time he would like to be able to roll back most changes that A may make to the system, except those specifically allowed by his policy.

To achieve this goal, the admin may regard his set of trusted servers as running in a global system pasture G, create a pasture A for the new application, and specify $G \rightarrow A$.

The arrow relation between two pastures can be limited to certain files, as described below.

3.2.2 The bidirectional arrow

The bidirectional arrow between pastures means that references to a resource, such as a file, by name from either pasture would return the same object. The applications running in these pastures can create the same races as in classical UNIX, and these would be resolved or allowed to exist as in classical UNIX.

With a bidirectional arrow relation qualified by a file name, applications running in two separate pastures can thus share a regular file, or communicate through a socket file by that name. An unqualified bidirectional arrow between pastures would mean that they are functionally identical in their access privileges.

3.3 Information flow implications

The information flow meaning of the unidirectional arrow $A \rightarrow B$ is as follows: the *writes* of processes in A are visible to those in B, but not vice versa. Changes to files made by processes in B and new files created by processes in B are not visible to those in A; B can freely read information from A, but its writes will not affect resources in A, since they result in copy-on-write of the resource written to.

The bidirectional arrow $A \leftrightarrow B$ (or, equivalently, a pair $B \rightarrow A$ and $A \rightarrow B$) means that information flow via