

Rationale and Design for a Pluggable Access Control Interface

Richard Offer
(offer@sgi.com)

May 30, 2001

Abstract

This paper presents a rationale along with a proposed design for a pluggable access control architecture for use in tandem with the Linux Security Modules currently under development.

1 Introduction

There seems to be a burgeoning of new security features and architectures that are, for the first time, reaching outside of the classic security community and into the main stream.

With the release of the NSA Security Enhanced Linux [SELinux] and the Role Set Access Control work [RSBAC] coupled with the traditional MLS systems that the TrustedBSD project [TrustedBSD] is targeting, the open source release of key components of a B1 system by SGI [OB1] and the alternative approaches such as Linux Intrusion Detection System [LIDS] and SubDomain [SubDomain], things have suddenly become more interesting, and more complex.

The main reason for this increase in complexity is not because of the variety of security models and policies, those have always been there, what is different, is that each of the above is open source. What does open source bring to the problem? Code sharing.

Previously high assurance operating systems were the preserve of commercial vendors that could afford, not only the resources to develop the software, but that found it easier to justify the increased merge costs when incorporating a new version, simply because that was done so

rarely.

While there were a number of independent projects underway that were attempting to add mandatory access control into the Linux kernel, none had yet been folded into the main source tree and hence were distributed as a set of patches. There is nothing inherently wrong with development by patches, it is a time honoured feature of Linux kernel development, but they are nevertheless a nightmare to handle, since the patches are against a vanilla kernel which will have no resemblance to that which the developer is using.

The problem finally came to head with a posting by Linus Torvalds who, when asked on a private mail list what the security goals were for 2.5 development replied.

...

The embedded people don't really want security management (or rather, to them, the simple "root is all" approach is fine, and takes up less space than more complex schemes with the selinux kind of security managers etc).

And even the security people aren't really sure which *sort* of security support they want (or rather - they all KNOW which kind of security management they want, but there are as many different opinions as there are people).

So I'm not interested in any one particular approach, TE/DTE/MLS or whatever. I can't even discuss the difference intelligently anyway (or even play at it), but what I can tell is that there is no "one right way". And that's without even getting into the issue of what the policy should be for them.

Which means that what *I* would require from something that gets integrated into Linux is either:

- true simplicity. “`uid == 0`” is this. capabilities are an approximation of this, and it turns out that almost nobody even uses capabilities just because they are complex enough to administer that they are of dubious value in many cases. The notion of an extended “`suid`” bit (with an ELF header of capabilities) has been bandied around for a long time, and the fact is that it would be a total maintenance nightmare. And that’s the *simple* case.

This is where Linux is now, and this is where we’ll remain, unless we get the alternative:

- truly generic. No “`MLS`” vs “`TE`” vs “`uid==0`” vs “`capability`” at ALL. Something where the “`uid == 0`” version of security is just one case (which the embedded people might use), or where SELinux would be just a matter of loading the SELinux module and installing *that* security model.

Quite frankly, nobody seems to be interested in actually trying to do the latter. Everybody has their *own* particular flavour that they want to push, and don’t realize that I cannot accept that as a maintainer.

...

Taking that as a call to arms, and a clear indication of what will and will-not be accepted into the the main-line Linux kernel, the Linux Security Module (LSM) [LSM] project was established.

But it seems to the author that everyone is missing the bigger picture. Once the LSM is in place, Linux will have the ability to change its security policy with something as simple as loading a kernel module. But what about the applications ?

Applications have security policies too, for example.

- `id` would like to display policy specific information (ie MAC label, capability information)
- `ls` needs to be able to display policy specific information such as any capabilities attached to a file (using extended attributes)
- `find` would like to be able to search for files that have certain policy specific information.
- `ps` would like to know the attributes for a process.

- `X` needs to know if it can allow cut-and-paste between two windows.
- `sendmail` has to be able to decide if an email message to a user is allowed given the corresponding security attributes of the message and user.

These examples of application access control are not fanciful, today Trusted Irix[Trix], a traditional MLS system has support for each of these. Any system that expects to meet the common criteria [CC] for a trusted system and be useful in the commercial environment (as opposed to a research project) will probably need simmiliar functionality.

Without some common mechanism for obtaining access control information, each one of these applications will need to be modified in a policy specific manner. This will lead to code-forking or code-bloat. Either the owners of the applications will refuse policy specific code (following Linus’s example) or it will be accepted and applications will have to code for every policy *on a run-time basis*. An application coded to use `mac_dominat_e()` is going to have problems if the MLS security module (implementing the `mac_*` system calls) is unloaded and a the TE module is loaded instead. Even if this was done at a reboot, there would still be problems. Would changing the policy mean that a whole new set of key applications would need to be installed ?

The problem is only heightened by the fact that the applications that are going to be most affected by a policy change are the very ones that are most security relevent.

2 A Solution

The same situation was present in a different part of security relevent application code a few years back. Every application was coded to perform its own authorisation. Then people wanted to start using alternative authorization policies than the classic ‘enter password’, they wanted ‘time of day’, biometrics and location policies that could be tailored to a site without changeing the application. They wanted a Plugable Authorization Mechanism.

Is Access Control, that much different to Authorization ? Yes, for a start its much harder. Accss control is more of a dialog than the simpler “if I’m not authorized then

exit” approach of most programs that require authorization. Developing a policy independent API is not going to be easy, however that shouldn’t stop us from developing the idea of Plugable Access Control Modules, aka PACM (pronounced pac’em).

2.1 Design Requirements

- Must be able to obtain security attributes for existing processes, files, sockets, and System V IPC.
- Must be able to modify process/file/socket/IPC attributes.
- Must be coded in C for maximum useability. Modules should be able to be coded in any language that is callable from C. Much as netfilter allows firewalls to be written in perl, there should be nothing to stop someone from using python to experiment with new security models.
- Must be thread safe.
- Applications should be able to be linked against both the PACM library and a given policy file statically, so that rescue disks can be built.

This will not be either the default or the recommended practise, but its important for certain key applications that it be possible.

- It should be possible to stack the modules and for information to be shared between co-operating modules.

The proposed architecture of PACM is heavily influenced by that of PAM, if for no other reason that it seems to work.

An application that uses PACM will link against a single library that is responsible for parsing the application specified configuration file and loading all the modules listed.

Applications specify a name that the library will then use to obtain the list of moduyles that will implement the policies. The name is the name of a configuration file that resides in the PACM configuration directory (/etc/pacm.d), the file has the following syntax.

type DSO args

Type is a simple means to limit which DSOs are used for a particular access decision, the following types are anticipated.

identity	this module returns identity relevent information.
privilege	this module return information regarding any privilege information.
cando	this module provides comparison functionality.

DSO is the full pathname to a PACM module, these will normally live in /lib/security.

Args are module specific arguments that a site can use to change the default behaviour of the module, these will be documented by the module in question.

On the initialzarion of the library by an application, a single function will be called (`pacm_init_module()`), this will initialize a structure with function pointers for the exact services that the module support. By adopting this design it supports the design goal of allowing policies to be linked into the application (defining well known function like PAM does will stop multiple modules being loaded into a single binary with “multiple references”).

2.2 The Goals of PACM

Cross-platform Most of the code that is likely to benefit from this framework is used on multiple operating systems, the framework should not be Operating system or architecture specific (polices can/will be). The side affect is that there must be a license firewall that allows both open source and commercial applications to use the code with no fear of contamination from any license that an individual module may be under. This means that all core code should be licensed under the BSD license.

It must be easy for applications to use The cost/benefit of adding PACM support to an application must be significant against the cost of maintaining separate code bases or writing application specific run-time selection of policy specific APIs. This implies the API is both easy to understand and fits in with current mechanisms, so applications do not need to be re-architected to use it.

It should be easy to write new modules LSM is going to spawn a new interest in security policies, it should be no harder to write the PACM module than it is to write the kernel module. If all the policy logic is implemented in the policy LSM, then the PACM module should need to do little more than call policy specific system calls.

Prove the concept PACM should ship with the modules that implement multiple LSM policies to prove it works, at a minimum, classic unix, capabilities, and one mandatory access module. Key applications should be ported to use PACM to prove the concept and act as the backdrop for a tutorial.

Stand on a shoulders of giants Reuse concepts, code and experience. PAM works and is familiar, use it as a model, but not religiously.

The design discussed in this proposal is heavily based on the PAM design and methodology; administrators construct a policy file (see 4.1.1 for an example), policy writers develop a shared library that supports their policy, application programmers write to the application API for the functionality they require.

It is debatable whether the need to wrap such Unix'isms as `uid`, `gid` or `groups` is needed, PAM decided not to, PACM proposes that we do, one reason is to allow the complete solution to be examined, but there is also the fact that existing applications do assume that a `uid` of 0 is special, with a policy that supported pure capabilities, this is no longer true.

3 Attributes

The design of PACM is based around the concept of security attributes, so there needs to be some way to obtain the attributes, set the attributes, compare attributes for any combination of subjects and objects (using MLS terminology).

Work is ongoing in specifying both the application and module APIs [PACM].

Applications use of security attributes can be broken down into three types.

1. those that only present security relevant information to users (`id`, `ls`, `ps`, etc.)

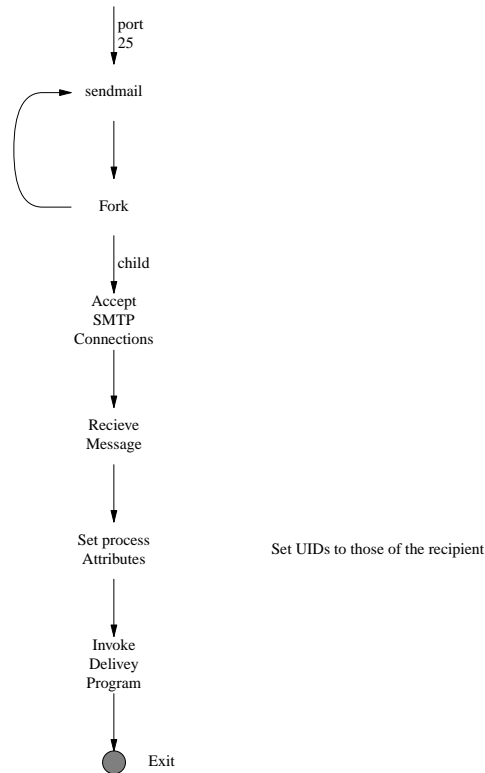


Figure 1: Simplified Control Flow of Sendmail

2. those that make decisions using system security relevent information (`sendmail`).
3. those that perform their own decisions using application specific security information (X).

Throughout this section we will consider the case of `sendmail`, which from a simple block diagram looks like figure 1.

3.1 Getting Attributes

From figure 2 it can be seen that the first place where attributes are needed is to determine those of the incoming message, by querying the attributes of the socket that has been opened. In traditional MLS systems with TSIX trusted networking, this would be as simple as a call to `tcprecvfrom()` followed by calls to

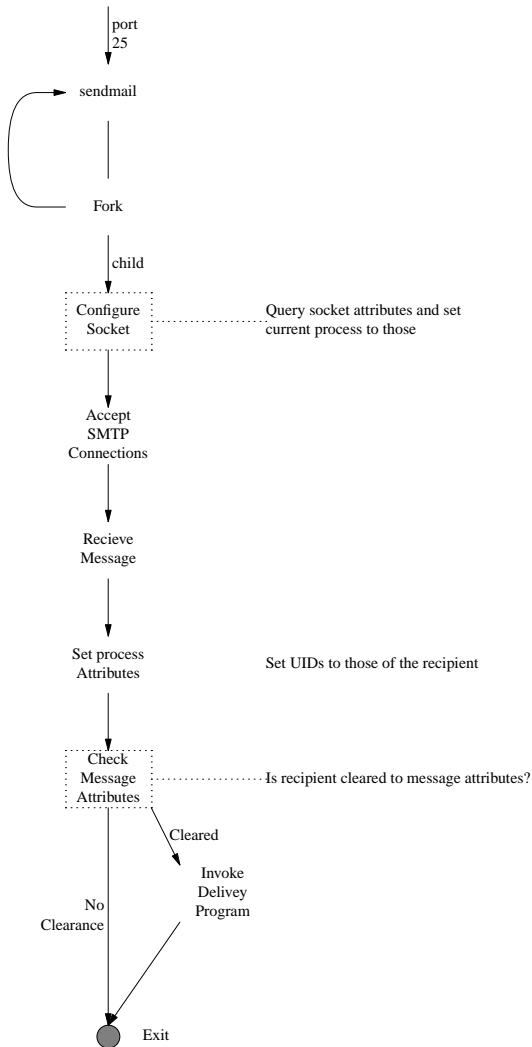


Figure 2: Simplified Access Control in Sendmail [Sendmail]

`t6get_attr(T6_CLEARANCE, ...)` to extract the MAC label for passing to `mac_set_proc()`.

Because TSIX is typically only found on traditional MLS systems with IPv4 and we want to allow future migration to IPsec, we would need replacements for this level of functionality.

3.1.1 Possible API

```

pacm_attr_t pacm_get_socket(pacm_t handle, int
sfd);
pacm_attr_t pacm_get_process(pacm_t handle, pid_t
pid);
  
```

3.2 Setting Attributes

Once we have the attributes of the socket, we need to change those of the process to match. We could explicitly set the attributes that we need, but we don't always know what those will be.

3.3 Cloning Attributes

Because of the inherent difficulty in specifying a generic attribute for anything interesting, the initial proposal attempts to solve the special case with the expectation that this will lead to a general solution at a later date. Rather than allowing applications to simply create an object/subject with a given attribute, we instead take a *selective breeding* approach. This requires more work, but it is at least achievable.

The crux of *selective breeding* stems from the observation that applications on the whole do not make up the labels (in MLS terminology) that they wish to use. They either obtain the information direct from user input (an example would be `find` asking the user for an MLS label to search for), or from some other subject/object. In the former case one can solve the problem by documentation, the `find` man page could simply say “`--sec-attr priv. priv` is a policy specific textual representation of an attribute to search for. See `pacm --help`”. Documentation of the actual textual format would be up to the policy to provide. A central clearing house for all policy documentation on the system could be obtained by using the hypothetical command, `pacm --help`.

In the case we are working through, `sendmail` should “merge” the attributes of the socket onto those of the current process. In this way the the application doesn't need to know about any policy specific types, it just needs to know that it want to take the `PACM_CLEARANCE` data from the socket and use it to set the current processes `PACM_CLEARANCE` level.

3.3.1 Possible API

```
pacm_attr_t pacm_clone(pacm_attr_t mother, long
mflags, pacm_attr_t father, long fflags);
```

3.4 Examining Attributes

Sendmail would probably like to record some details on what it is doing, this is at least the default case on many Linux distributions. The file `/var/log/maillog` records status about the messages passing through the machine. On a trusted system it should probably record the clearance information as part of the status.

For that to happen we need a way to extract policy specific information from the attributes. This is the identical to the requirements for programs such as `ls`, `id` etc.

A policy module should provide a human readable textual output representation for any attribute requested, if it is at all possible they should also make available an integer format representation (for example, capabilities could be represented as a integer value, mac labels not).

If the attribute in question is a list of values, an array representation should also be available for the application to query, but the single textual representation must still be available.

An example is that if the textual representation for `PACM_INDIVIDUAL` is requested, then the username should be returned, if it is requested in integer format, then the `uid` should be the value.

3.4.1 Possible API

```
int pacm_get_attr(pacm_attr_t subj, long what,
const char *fmt, ...);
```

3.5 Making Decisions

In a traditional MLS system, desions are easy, its either `mac_dominant()` or `mac_equal()`. No information regarding what the decision will be used for is used in determining the result and it is up to the application to implement its own policy by deciding which to use. This is a flaw in an generic access control mechanism, policies may want to handle (for example) read, write and create decisions differently.

So, what we need from the PACM api is tha ability to pass additional information to the decision making func-

tion that can optionally be used to determine whether the access is allowed.

To allow for the cases where insufficient attributes are set on the either the subject or object, Casey Schaufler [Schaufler] has suggested the return value should not be a simple boolean value. For example, consider the case where the capability set is non-existant (which is different to it being empty). If a access decision would require a capability to succeed, rather than return `NO`, perhaps `UNLIKELY` would be a better response?

```
int pacm_cando(pacm_attr_t subj, pacm_attr_t obj,
pacm_action_t what);
```

4 Other Applications

4.1 id

The application `id` is simple, at its basic, it reports the invokers user `id`, their group `id`, and any supplementary groups that they belong to.

On Irix, additional command line flags are allowed to display the capability set (`-P`) and the mac label (`-M`) of the invoking user.

4.1.1 Configuration File

```
# id pacm configuration file for a POSIX MLS system
#
# _unix returns INDIVIDUAL, FAMILY and CLUB
identity /lib/security/pacm_unix.so
#
# returns ROLE
identity /lib/security/pacm_posix_mac.so
#
# returns IMPORTANCE
privilege /lib/security/pacm_posix_capabilities.so
```

4.2 x

`X` is an interesting case to review, it is the only case so far considered that needs to perform access control on application data. It is not surprising that SELinux never provided a trusted version of `X`, but left that as a future project. The access controls are not determined by `uid` or `gid`, but on the `X` display connection. This of course requires that we be on a system with trusted networking.

4.2.1 Configuration File

```
# x11 pacm configuration file for a CMW system
#
# _t6 returns INDIVIDUAL, FAMILY and CLUB,
# ROLE but only for trusted network sockets.
identity /lib/security/pacm_t6.so
#
# returns IMPORTANCE
privilege /lib/security/pacm_t6.so
#
# uses SGI Access CMW extension
cando /lib/security/pacm_sgi_access.so
```

For a normal (non-CMW) approach, all that should be needed is a null file.

```
# x11 pacm configuration file for a classic X11 system
#
# Nothing here, we're using default X11 access control
# (once a display is opened, everything is permitted)
#
```

5 Audit

Where does Audit fit into PACM ? If possible, audit should be contained in a PACM module, for the same reason as PACM, we don't want to fork applications just because audit is a requirement for some cases.

If there is enough information to make an access control decision, then there should be enough to generate an audit record.

6 Alternative Approaches

An alternative approach suggest by Casey Schaufler [Schaufler] would be to extend the TSIX trusted networking api to support non-sockets types. While the approach does have considerable merit in not re-inventing the wheel, the major problem is that the design of TSIX stems from an MLS approach where the types of the data and attribute types are both known by the application.

Another approach would be for an application to code to an existing standard (POSIX 1e) and for a library to perform "concept" lookup, so that an application could reference "dblow" and that would be mapped to an equivalent meaning under the current policy. The problem with that is that the mapping would be site specific.

A final possibility would be to use a little language (XML) so that an application could define its exact requirements for processing, and then for a library to in-

terpret those requirements and return another XML document with the results.

7 Conclusion

This document outlines a proposed solution for the design and implementation of a Pluggable Access Control Mechanism. A number of issues still stand, is it suitable for all policy types? The authors experience lies within the MLS environment, but if this is going to work all policies must be embraced equally.

Input is sought on any cross-policy issues that need to be resolved.

References

- [CC] <http://www.commoncriteria.org/>
- [LIDS] <http://www.lids.org/>
- [LSM] <http://lsm.immunix.org/>
- [OB1] <http://oss.sgi.com/projects/obl/>
- [PACM] Richard Offer, *Pluggable Access Control Mechanism: API Specification*, 2001 (in progress)
- [PAM] <http://www.kernel.org/pub/linux/libs/pam/>
- [RSBAC] <http://www.rsbac.org/>
- [Schaufler] Private correspondence.
- [SELinux] <http://www.nsa.gov/selinux/>
- [Sendmail] SGI Internal Document, The Irix/Trusted Irix Sendmail Specification.
- [SubDomain] <http://www.immunix.org/subdomain.html>
- [Trix] <http://www.sgi.com/>
- [TrustedBSD] <http://www.trustedbsd.org/>