Policy Analysis for Security-Enhanced Linux

Beata Sarna-Starosta and Scott D. Stoller

Computer Science Department

State University of New York at Stony Brook

http://www.cs.sunysb.edu/~stoller/

Security-Enhanced Linux (SELinux)

SELinux = Linux plus a kernel module that enforces security policies expressed in SELinux policy language.

Implements **mandatory access control**: security administrator imposes an access-control policy that other users cannot modify.

Example: Apache configuration file can be modified only by Apache executables executed by the user Apache Administrator.

Example: finger daemon can modify only a specified log file.

Traditional **discretionary access controls** in UNIX are inadequate for highly secure systems.

The Outlook for SELinux

Run-time overhead of policy enforcement is low (a few per cent).

SELinux is going **mainstream**: it is incorporated in version 2.6 of the Linux kernel.

The **difficulty of policy development** is a significant hurdle to widespread use of SELinux.

SELinux developers created an **example policy**. It is **large** and **low-level**. Users will need to **combine and customize** policies for different applications and services.

Determining whether an SELinux policy meets **high-level security goals** is **hard**.

Outline

Overview of PAL: Policy Analysis using Logic-Programming

Benefits of PAL

Related Work

Model of SELinux Policies

Information Flow

Queries

Future Work

Our Approach to SELinux Policy Analysis

PAL: Policy Analysis using Logic-Programming

Goal: Help user determine whether an SELinux policy meets high-level security goals.

Methodology:

1. Automatically translate SELinux policy into a logic program.

2. Use simple logic programs to query (ask questions about) the policy.

Benefits of PAL: Flexibility

Logic programs are more flexible and expressive than typical special-purpose query languages.

Novice users can use a **library** of query templates.

Intermediate users can also create **simple variants** of queries in library.

Advanced users can also **develop new queries quickly** in the high-level, mostly-declarative logic-programming language.

Benefits of PAL: Efficiency

PAL is implemented in **XSB**, an efficient tabulation-based logic programming system developed at SUNY at Stony Brook.

XSB's **goal-directed** query evaluation avoids analyzing irrelevant parts of the query.

Our translation of the policy **preserves** (doesn't expand) **macros**, which are used extensively as a form of abstraction in SELinux policies. In effect, macros are expanded **on-demand** during policy analysis.

We translated the original policy (yielding 8.4 KLOC) and the macro-expanded policy (yielding 23 KLOC) and ran queries on both, to compare performance.

Benefits of PAL: Justification

XSB's **justifier** can help explain query results to the user.
The justifier displays computation paths that led to the result.

Example: Consider information-flow property "All information flow from X to Y passes through Z."

If the policy violates this property, the justifier shows the user one or all computation paths that correspond to counterexamples.

If the policy satisfies this property, the justifier shows the user computation paths that correspond to all information-flow paths from X to Y, so the user can see that they all pass through Z.

We have not yet integrated the justifier and PAL.

Related Work: SLAT

SLAT: Security-Enhanced Linux Analysis Tools

SLAT is from J. Guttman, A. Herzog, and J. Ramsdell at MITRE.

PAL adopts SLAT's model of information flow.

SLAT has a special-purpose regular-expression-based language for expressing information flow properties.

SLAT gives "yes" or "no" answers. A "no" answer is accompanied by one counterexample.

PAL can show all counterexamples and can give sets, relations, etc., as answers.
Example: Find all X such that information can flow from X to Y without passing through Z.

Related Work: Apol and Gokyo

Apol, from Tresys Technology, provides a policy browser and graphical display of a type transition graph and an information-flow graph.

Apol does not support a language for expressing properties or queries.

The Gokyo project at IBM Watson focuses on identifying and achieving specific policy design goals, including integrity of the TCB and completeness of the policy.

Gokyo's policy analysis tool supports this by manipulating and graphically displaying sets of permissions.

Gokyo does not analyze (transitive) information flow.

SELinux Security Contexts

security context: security-relevant information about a **resource** (file, process, socket, etc.), specifically, [*Type, Role, User*].

user: similar to ordinary UNIX notion of user. Tracked separately by the SELinux module.

role: A user (e.g., a system administrator) may act in different roles, depending on the task at hand. A "dummy" role, `object_r`, is used for resources that are not processes.

type: set of resources with the same access-control requirements

Example: files whose names match `/dev/hd*` or `/dev/sd*` have type `fixed_disk_device_t`. Processes running executables used for filesystem administration have type `fsadm_t`.

SELinux Policies

policy: a sequence of rules that define several relations.

`role`(R, T): a process with role R may have type T.

`user`(U, R): a process of user U may enter role R.

A security context [T,R,U] is **consistent** if
(1) `role`(R, T) and `user`(U, R) hold, or
(2) R is `object_r` and T is not a process type.
Inconsistent security contexts are prohibited at run-time.

Permissions are associated with types (see next slide). Thus, the `role` and `user` relations indirectly (through types) limit the permissions available to a process with a given role and user.

SELinux Policies (cont'd)

`type`(*T*, *Aliases*, *Attributes*). Attributes are used in other rules to represent the set of types with that attribute.

`access_vector(allow,` *SourceT*, *TargetT*`, Class, Perm)`: resources (typically processes) with type *SourceT* are allowed to perform operation *Perm* on resources with class *Class* and type *TargetT*.

class: file, directory, process, socket, etc.
permission: read, unlink, signal, sendto, etc.

`role_allow(`*RoleSet*, *NextRoleSet*`)`: a process with a role in *Role-Set* is allowed to transition to roles in *NextRoleSet*.

macros are used to define names for sets of classes, permissions, and rules.

Information Flow Graph [SLAT]

Nodes: consistent security contexts.

Edges: [*T1*, *R1*, *U1*] $\xrightarrow{[C,P]}$ [*T2*, *R2*, *U2*] if
(1) a resource with security context [*T1*, *R1*, *U1*] is allowed to perform a **write-like** operation P on a resource with class C and type $T2$, or
(2) a resource with security context [*T2*, *R2*, *U2*] is allowed to perform a **read-like** operation P on a resource with class C and type $T1$.

Declarations in the policy indicate which operations are **read-like** and which are **write-like**.

Queries: Information Flow

Information-flow queries ask about paths in the information-flow graph. We formulate such queries as logic programs representing automata that accept the paths of interest.

Example from SLAT: Check whether the policy adequately restricts access to raw disk data, i.e., accesses to `fixed_disk_device_t`.

Hypothesis: information flow from a standard user's security context to `fixed_disk_device_t` may occur only if the information passes through the filesystem administrator type `fsadm_t`.

SLAT shows 1 counter-example. PAL shows it and a few others. Running time: 0.9 sec. With macro-expanded policy: 2.3 sec.

Details of `fixed_disk_device_t` Info-Flow Query

Check whether there is an information-flow path from a security context satisfying $T = \texttt{user\_t} \wedge R = \texttt{user\_r} \wedge U \neq \texttt{jadmin}$ to a security context satisfying $T = \texttt{fixed\_disk\_device\_t}$ that does not pass through a security context satisfying $T = \texttt{fsadm\_t}$.

```
init(fdisk_aut, [user_t,user_r,U], [neq(U,jadmin)]).
trans(fdisk_aut, [T0,R0,U0], (C,P), [T1,R1,U1], [neq(T1,fsadm_t)]).
trans(fdisk_aut, [T0,R0,U0], (C,P), [fixed_disk_device_t,R1,U1], []).
final(fdisk_aut, [fixed_disk_device_t,_R,_U], []).
```

Queries: Information Flow 2

Find all security contexts from which information can flow into `shadow_t` (which contains sensitive authentication information).

PAL finds 53 such security contexts.
Running time: 1.0 sec. With macro-expanded policy: 2.7 sec.

This query returns a set and hence cannot be done with SLAT.

```
transitive_flow(X,Y) :- flow_trans(X,Y).
transitive_flow(X,Y) :- flow_trans(X,Z), transitive_flow(Z,Y).
transitive_flow(SourceContext, [shadow_t, Role, User]).
```

Queries: Integrity

Check integrity of Jaeger *et al.*'s initial proposal for a Trusted Computing Base (TCB) for SELinux, expressed as a set of trusted types. In other words, find types outside the TCB from which information can flow into the TCB in 1 step.

PAL, like Gokyo, reports numerous potential violations.
Running time: 0.3 sec. With macro-expanded policy: 1.1 sec.

```
integrity_violation(TCB, TCBType, OutType) :-
flow_trans([OutType,OutRole,OutUser], Class, Perm, [TCBType,TCBRole,TCBUser]),
member_type(TCBType,TCB), not_member_type(OutType,TCB).
```

Queries: Separation of Duty

"Perhaps the most basic separation of duty rule is that any person permitted to create or certify a well-formed transaction may not be permitted to execute it" [Clark and Wilson, 1987].

Find all file types for which some daemon type has execute permission and a write-like permission.

PAL indicates that there are none in less than a second.

This implies that compromised daemons cannot infect executables, foiling RootKit and similar attacks.

Queries: Completeness

`neverallow` rules in a policy have the same format as `allow` rules but the opposite meaning. Policy compiler checks for consistency of the `allow` and `neverallow` rules.

Find permissions that are neither allowed nor explicitly prohibited by the policy. Gokyo pointed out that such permissions reflect a kind of incompleteness in the policy.

This query is easy to do with PAL or Gokyo.

```
unspecified_permissions(SrcType,TargetType,Class,Perm) :-
  is_type(SrcType), is_type(TargetType),
  perm_valid_for_class(Perm,Class),
  \+ access_vector(SrcType,TargetType,Class,Perm),
  \+ neverallow1(SrcType,TargetType,Class,Perm).
```

Future Work

Policy Analysis using Logic-Programming is **flexible** and **efficient**, but there is still work to do:

incremental policy analysis, as the policy changes.

compositional policy analysis, since policy fragments associated with different applications may interact.

implementation and analysis of **trust management** policies.