Madison: A New Approach to Policy Generation

Karl MacMillan
Red Hat
kmacmill@redhat.com

Abstract

This paper introduces a new library and associated tools, called Madison, for automatic policy generation. Madison includes features that address all aspects of fully automated and user guided policy generation, but has a particular focus on the generation of policy statements similar in form to hand-written reference policy modules. This focus allows Madison to compliment the existing tools that perform automatic policy generation. The policy modules generated by Madison use reference policy interface calls and allow rules in accordance with the encapsulation rules of the reference policy. Producing these interface calls requires infrastructure to parse reference policy headers, analysis routines to determine the access allowed by each interface, and a approach to optimizing the set of interface calls for a module that is computationally tractable.

1 Introduction and Motivation

Recent features added to the SELinux tools and kernel mechanism allow convenient customization of the policy without requiring policy development. Conditional policy extensions, loadable policy modules, the reference policy (2), and semanage (1) have introduced runtime and development time options to the policy and, in the case of loadable policy modules, allowed the convenient distribution of policy beyond the base policy included with the operating system. This increases the configurability of SELinux to the point that it can be readily adapted to many systems.

However, this increase in the configurability has not obviated the need for writing policy. There will always be a need to write policy for custom or modified applications or to conform to unusual requirements and configurations. The opportunity to create custom policy is one of the strengths of SELinux. However, writing policy by hand remains a task for SELinux experts.

Automated policy generation is an opportunity to bring the benefits of custom policy to those without deep SELinux expertise. It can also be used to automate some of the more tedious tasks for experienced policy developers. In the broadest sense, automated policy generation determines the access required by one or more applications and generates appropriate SELinux policy statements to allow that access. The process can be fully automated or be guided by a human operator.

Within that broad definition of automated policy generation there are many sub-problems. For example, determining the required access can involve simple or complex observation of running software, analysis of application binaries, or reasoning about whether accesses requested by an application are necessary and secure. Similarly, generating policy from a set of required accesses can involve the mapping of processes and files to new or existing types, generating basic allow rules, and calling M4 macros.

This paper introduces a new automated policy generation library and associated tools, called Madison. Madison, written in Python, includes features for all aspects of policy generation, but has a particular focus on policy statement generation. It attempts to:

- Generate reference policy style policy modules similar to hand-written policy.
- Automatically extract information from reference policy interface files to allow the generation of interface and template calls from accesses.
- Enable guided policy development by providing users with alternative interface calls and warnings about high-risk accesses.
- Enable iterative policy development by intelligently adding access to existing hand-written and generated policy modules.

- Include generation of new policy modules based on a related, or parent policy module. For example, generating policy modules for CGI scripts based on an existing policy for Apache.

This paper will focus on the interface generation mechanism in Madison. First we will give a brief overview of the reference policy, briefly examine other policy development tools, and introduce the basic design of Madison.

2 Reference Policy Overview

The reference policy is a relatively new system policy that replaces the example policy initially included with the SELinux. The reference policy includes several important new features including strong modularity, support for multiple policy variants from a single source tree, compatibility with loadable policy modules, and a structured development environment.

From a policy generation perspective, the strong modularity and structured development environment provide an opportunity to generate better and more secure policy modules. The modularity enforces encapsulation, only allowing types, attributes, and other identifiers to be directly used in the module in which they were defined. Access to another modules' private resources is provided through interfaces.

For example, consider the following policy statements taken from the Apache policy included with the reference policy:

```
allow httpd_t self:fd use;
allow httpd_t self:sock_file r_file_perms;
allow httpd_t self:fifo_file rw_file_perms;
allow httpd_t self:shm create_shm_perms;

dev_read_sysfs(httpd_t)
dev_read_rand(httpd_t)
dev_read_urand(httpd_t)
dev_rw_crypto(httpd_t)

fs_getattr_all_fs(httpd_t)
fs_search_auto_mountpoints(httpd_t)

term_dontaudit_use_console(httpd_t)

auth_use_nsswitch(httpd_t)

corecmd_exec_bin(httpd_t)
corecmd_exec_sbin(httpd_t)
```

The allow rules at the beginning of this example all refer to types declared within the Apache policy module. The interface calls (e.g, `dev_read_sysfs(httpd_t)`) allow access to types declared in other modules.

The interface calls are defined in separate files that are installed, along with additional support macros, as part of a development package for the reference policy. The interface and support macro files are referred to as headers, in keeping with C/C++ development ideas. The installation of the headers allows third-party modules not distributed with the reference policy to use these interfaces during compilation.

The use of interfaces in the reference policy increases the readability and structure of the policy. It also allows a module author to define what access is appropriate to the resources declared within a module. For this reason, we feel that it is important for automated policy generation tools use these interfaces. It gives the automated policy generation tool a wealth of information about what access is expected and what access is potentially dangerous.

3 Prior Work

The goal of Madison is not to reproduce existing automated policy generation tools and research, but rather compliment them. It is envisioned that in the future it may be possible to integrate the functionality of Madison and other tools.

3.1 audit2allow

The command-line tool audit2allow is a simple and widely used tool for converting audit messages into policy. As its name suggests, it was originally used to convert audit messages into basic allow rules. Today it can also generate modular policy require statements and reference policy interface calls, though these features have several serious limitations. It has no provision for the creation of types, relying on other tools or the policy developer to declare types and provide enough policy to allow those types to be applied to processes and objects so that audit messages can be generated.

Madison is, in many ways, a direct descendant of audit2allow. The code base began with the latest version of audit2allow, though significant changes have been made. The goals of Madison are a superset of those for audit2allow.

3.2 Polgen

Polgen (3) is an ambitious project from Mitre for automatic policy generation. In contrast to Madison, it is focused on determining the access required by applications including recognizing common access patterns and automatically generating types. It includes a modified version of strace that allows it to observe the system calls made by a process or set of related processes. It then derives types and common access patterns from the syscall

information to generate both type declarations and allow access.

Polgen was started before the reference policy uses a small set of custom M4 macros to allow access instead of interface calls. Work is ongoing to generate reference policy interface statements instead.

Polgen and Madison are largely complimentary. It is imagined that the type declaration and access requirements from Polgen could be used to drive the output capabilities of Madison, though no concrete work has been done in this direction.

3.3 Slide

Slide (http://oss.tresys.com/projects/slide) is a development environment for the reference policy based on the popular Eclipse framework. In addition to standard editing and project management capabilities provided by most IDEs, SLIDE includes some basic policy generation and search capabilities. This includes the ability to generate standard access patterns on policy module creation (e.g., allow access commonly needed by a network facing daemon) and generating interfaces and other low-level policy patterns during editing. The search capabilities allow searching interfaces for relevant access based on types, object classes, and permissions.

Recent versions of Slide also provide an audit viewing capability. Private discussions with the developers indicate that this capability is expected to include the ability to generate policy from the audit logs in the future.

Again, Madison can be seen as complimentary to Slide. It is imagined that Slide could use the policy generation and interface searching capabilities of Madison in the future.

4 Overview of Madison

Madison is composed of a Python library and a set of tools based on that library. The library currently contains components for:

- Audit message parsing from either syslog or the audit daemon.
- Reference policy header parsing.
- Data structures for representing SELinux policies, reference policy headers, and more abstract policy concepts needed for policy generation .
- Analysis algorithms for converting reference policy headers into a representation suitable for policy generation.
- Policy generation algorithms that can generate simple SELinux policy rules and reference policy interface calls.

The library also includes a comprehensive test suite and documentation.

Significant effort is made to keep the components of the library loosely coupled. This should allow, for example, for an application to drive the policy generation components without using the audit parsing routines.

The primary tool currently included with the library is `audit2policy`, which is designed to serve as a drop-in replacement for `audit2allow`, though with better support for the reference policy policy. Gui tools for interactive policy development are planned but have not yet been implemented.

5 Generating Reference Policy Modules

As was previously mentioned, the primary focus of Madison is generating reference policy modules. This section will detail how that is accomplished. We will largely limit the discussion to the type enforcement policy. It is planned that user and role authorizations, MLS/MCS, and constraints will be addressed in later versions of Madison. However, type enforcement is expected to always form the bulk of the generated policy.

Madison attempts to choose reference policy interfaces for requested access based primarily upon information extracted from the reference policy headers and the linked binary policy modules. Prior to choosing an automated approach, we considered manually deriving the access allowed by each interface and including that information in XML comments included with each interface. The automated approach was chosen because of the large number of interfaces present in the reference policy and the maintenance burden imposed by maintaining the same access information in the interface body and comments.

5.1 Generation Process Overview

The policy generation components in Madison are based on two primary concepts adapted from the SELinux enforcement mechanism: access vectors and access vector sets. Access vectors are the smallest allowable unit of access within SELinux. They represent access between a subject and object, represented by a type or a full security context, an object class, and one or more permissions. Within Madison, access vectors currently represent the subject and object as types only.

Access vector sets contain one or more non-overlapping access vectors. An access vector set is nothing more than a sparse representation of a full access ma-

trix. The sparse representation allows the efficient storage and searching of the access vectors. The full access allowed by a type enforcement policy is representable as an access vector set; The avtab - or access vector table - in the SELinux security server is one implementation of an access vector set.

Within the type enforcement policy, all access can be reduced to an access vector or access vector set. For example, a simple allow rule is direct expression of a single access vector:

```
allow passwd_t shadow_t : file read;
```

More complex allow rules can be expanded into an access vector set. It is also possible to convert other policy statements, e.g., typeattribute statements, combined with allow rules into an access vector set.

Madison uses these concepts as the building block for policy generation. Requested access, used as input to the policy generation component, and access allowed by a reference policy interfaces can both be expressed as access vector sets. This representation allows the use of simple set operations (e.g, union, subset, and difference) for matching requested access to interfaces and the ordering of reference policy interfaces. A discussion of the properties of the reference policy interfaces that make ordering using set operations practical can be found later in the paper.

The basic work flow for generating policy using these basic concepts is as follows:

1. Parse the reference policy headers into internal data structures.
2. For each interface in the reference policy headers, generate an access vector set representing the access allowed by that interface.
3. For each access vector in the requested access, find the interfaces whose access vector set is a superset of the requested access vector. For requested access vectors that do not match any interfaces, allow rules can optionally be generated in the final step. Encode access allowed by the interface not in the requested access as a numeric distance.
4. Within the list of interfaces matching each a requested access, order the interfaces by the distance. Choose the interface that allows the least access.
5. Reduce the output list of interfaces by removing those interfaces whose access is a subset of another interface.
6. Generate policy source by calling each interface from the optimal set and any allow rules and require statements for non-matching accesses.

Each of these steps is examined in greater detail in the following sections.

5.2 Parsing Reference Policy Headers

Madison includes a newly built parser capable of parsing all of the reference policy headers. It understands the limited subset of M4 used in the reference policy in addition to the SELinux policy language. The output of the parser is an intermediate representation (IR) that is suitable for further analysis.

The decision to create a new parser rather than expanding an existing parser, like the one used in `checkpolicy`, was based on the two primary factors. First, the currently available parsers lack an IR representation that preserves sufficient syntactic information to allow policy generation. Second, the current reference policy syntax is slated for replacement by native SELinux policy language constructs.

It was felt that the creation of a new parser in Python would allow rapid prototyping of an IR format suitable for both policy generation and policy compilation while allowing Madison to make progress quickly. We hope that the current parser in Madison can be retired when the new policy language constructs are completed. However, this will require the creation of a new IR representation within the `checkpolicy` compiler with some similar properties to the one found in Madison.

To further understand the need for a new parser, let us examine the reference policy headers in more detail.

Parsing the reference policy headers involves parsing the subset of M4 syntax used by the reference policy, the SELinux policy language, and the intersection of the two. Listing 1 shows an example of a typical interface header. In this example you can see:

- XML formatted comments that contain both human readable documentation and machine readable information about parameters.
- Modular policy require statements generated through the `gen_require` reference policy macro.
- The use of the interface key word to define an M4 macro. This keyword, which is reference policy specific, is an alias for the more general concept of an M4 macro with arguments.
- M4 macro arguments - `$1` in this example. M4 macros can accept any number of string arguments that are referenced in the body of the macro by `$N` where `N` is the position of the argument.
- Interface calls - `files_search_usr` in this example.

- SELinux policy with interface arguments (`$1`) and permission set macros (e.g., `r_dir_perms`.

Listing 1 A basic reference policy interface

```
########################################
## <summary>
##      Read man pages
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary>
## </param>
## <rolecap/>
#
interface(`miscfiles_read_man_pages',`
  gen_require(`
    type man_t;
  ')

  files_search_usr($1)
  allow $1 man_t:dir r_dir_perms;
  allow $1 man_t:file r_file_perms;
  allow $1 man_t:lnk_file r_file_perms;
')
```

Listing 2 shows a more complex example. This example includes deep nesting of M4 ifdefs combined with SELinux policy language conditional statements.

Listing 2 A more complex reference policy interface

```
ifdef(`distro_rhel4',`
term_relabel_all_user_ttys($1_su_t)
term_relabel_all_user_ptys($1_su_t)
ifdef(`strict_policy',`
  if(secure_mode) {
   userdom_spec_domtrans_unprv_users($1_su_t)
  } else {
   userdom_spec_domtrans_all_users($1_su_t)
  }
')
')
```

Finally, Listing 3 shows several permission set definitions defined as part of the support macros. The support macros are primarily M4 macro definitions. The macros shown in this listing are an example of macros without arguments. Any text matching the macro name will be replaced with the text in the body of the macro. The replacement is recursive so the body of a macro can refer to another macro previously defined.

The deep nesting of policy statements shown in Listing 2 highlights one of the largest drawbacks to the `checkpolicy` parser. The original SELinux policy language had no concept of scoping or nested blocks. When the conditional policy language and optional blocks were added to the language, the parser was modified to have a limited notion of scoping, but retained

Listing 3 Several permission set M4 macros

```
#
# Permissions for getting file attributes.
#
define(`stat_file_perms', `{ getattr }')

#
# Permissions for executing files.
#
define(`x_file_perms', `{ getattr execute }')
```

many of the existing limitations. The lack of a generic tree-based IR makes adding additional nesting difficult. This is one of the reasons that conditional policy blocks cannot be nested.

Additionally, the `checkpolicy` parser has traditionally converted syntactic information into a semantic representation as soon as possible. For example, identifiers are hashed and replaced by integer values during parsing. The modular policy work relaxed this practice somewhat. For example, evaluation of special type operators like `self` or `-` is now deferred to module expansion where it used to be done immediately in the parser. However, it would require large changes to retain enough syntactic information to represent the unexpanded M4 macros. Most importantly, it is necessary to retain the macro parameters, e.g., `$1`, unchanged.

Contrastingly, the IR added in the Madison parser adds a significant degree of flexibility. It is loosely an abstract syntax tree (AST) commonly used in compilers. The tree structure easily represents the deep nesting of policy constructs found in the reference policy headers and preserves the syntactic elements Madison needs to correctly interpret the interfaces.

Additionally, within the IR, all identifiers are preserved as strings. This allows M4 macros and macro parameters to interpreted later by other parts of Madison. This also delays the processing of special identifiers like `self` and and `-`. While this delayed interpretation adds complexity to the analysis components it adds needed flexibility.

5.3 Determining Access

The IR output from the parser is further analyzed to create an access vector set for each interface. Madison retains interface parameters as strings during this analysis, resulting in access vector sets that reference `$N` parameters. The searching and generation components correctly interpret these strings.

Much of this analysis is simply expanding allow rules and recursively expanding interface class. For example, the interface in Listing 1 contains only allow rules and an interface call.

Other interfaces allow access more indirectly, sometimes requiring information not available from the interfaces. For example, consider the following interface:

```
interface(`dev_rename_all_blk_files',`
  gen_require(`
    attribute device_node;
  ')

  allow $1 device_t:dir rw_dir_perms;
  allow $1 device_node:blk_file rename;
')
```

The interface above allows the passed in domain type access to the attribute `device_node`. Correctly matching a requested access to a file that has the `device_node` attribute requires expanding the attribute, either during searching or analysis of the IR. Unfortunately, the expansion requires information that is not necessarily contained in the reference policy headers. This can be handled by extracting a mapping of attributes to types from the linked binary policy.

Interfaces that apply attributes are more challenging. For example, consider the following interface:

```
interface(`auth_can_read_shadow_passwords',`
  gen_require(`
    attribute can_read_shadow_passwords;
  ')

  typeattribute $1 can_read_shadow_passwords;
')
```

In the interface above all of the access is allowed through adding the attribute `can_read_shadow_passwords` to the passed in domain type. Expanding this access requires all of the allow rules that reference the `can_read_shadow_passwords` attribute, which are likely in the .te files of the modules rather than the headers. Again, this information can be extracted from the linked binary policy, but it requires loading and performing relatively sophisticated analysis of the loaded policy.

5.4 Interface Matching and Optimization

Once the interfaces are expanded into access vector sets, matching requested access in the form of access vectors is straightforward. Each interface access vector set which is a superset of the requested access vector is a match. This will, typically, yield several matching interfaces for each input access vector. Choosing among the multiple matches is more challenging, particularly if a global optimization across all of the requested access is attempted.

The general problem of matching N access vectors to an optimal set of access vector sets representing interfaces, where the optimal set means the fewest number of interface calls that allows as closely as possible only the requested access, is a complex problem. It appears to be an instance of the hitting set or set cover problem, both of which are NP-hard in the simplest case (the analysis of this problem was originally performed by the Setools team at Tresys, including Jason Tang, Kevin Carr, and Jeremy Mowery and communicated privately to the author). This makes the brute force solution too computationally complex given the number of interfaces and potential number of inputs.

This general form of the problem statement assumes, however, that the access vector sets representing the interfaces are randomly distributed subsets of the total access allowed by the policy. It is clear, however, that the encapsulation rules in the reference policy and the general avoidance of allowing unrelated access in a single interface means that the subsets not randomly distributed. More specifically, interfaces tend to either not overlap (e.g., `dev_read_framebuffer` and `libs_use_shared_libs`) or to form strict superset relationships (e.g., `dev_rw_framebuffer` is a superset of `dev_read_framebuffer`). Initial analysis of the reference policy interfaces shows these properties to generally hold for most interfaces, though more detailed analysis is in progress.

Assuming these properties, a much simpler optimization strategy can yield good output. In fact, the global optimization can be dropped entirely. The algorithm currently used in Madison is as follows. Each requested access vector is matched to the interface that has the smallest difference without consideration of the other requested access vectors. The interface list output from the individual matching is reduced by removing interfaces that are a subset of another interface present in the list.

Initial results using this algorithm are generally promising, with the interfaces chosen generally matching what a human policy writer might produce. This algorithm has the added advantage of being very low complexity. Further optimization is possible by indexing the interface access vector sets to avoid linear searching. Recent work has also introduce weighting into the set difference algorithm. This uses information flow weighting on the permissions (e.g., an interface that allows getattr and search for a directory in addition to the requested access should likely be chosen over one that allows write).

6 Future Work

Madison is in the early stages of development. The next area of development will focus on evaluating the security risk posed by the generated interface calls and presenting the users with alternatives. This work will likely include the introduction of additional tags in the interface XML documentation to allow interface authors to

indicate high-risk or seldom used interfaces and suggest alternatives.

Work on graphical tools for policy generation is also anticipated. These tools will focus on an iterative, human guided generation process. Initially these tools will focus on basic customizations likely to be done by administrators rather than more complex development by application authors. This will also include integration with existing graphical tools like Setroubleshoot.

References

[1] MACMILLAN, K., BRINDLE, J., MAYER, F., AND TANG, J. Design and implementatino of the selinux policy management server. In *Proceedings of the 2nd Annual SELinux Symposium* (2006).

[2] PEBENITO, C. J., , MAYER, F., AND MACMILLAN, K. Reference policy for security enhanced linux. In *Proceedings of the 2nd Annual SELinux Symposium* (2006).

[3] SNIFFEN, B. T., HARRIS, D. R., AND RAMSDELL, J. D. Guided policy generation for application authors. In *Proceedings of the 2nd Annual SELinux Symposium* (2006).