Approved for Public Release; Distribution Unlimited
Case #06-0046

Guided Policy Generation for Application Authors

Brian T. Sniffen David R. Harris John D. Ramsdell
*The MITRE Corporation**

Abstract

Polgen is a tool for human-guided semi-automated SE Linux security policy generation. Polgen processes traces of the dynamic behavior of a target program. In that behavior, it observes instances of information flow patterns such as Pipeline, Interpreter, and Proxy. Based on the patterns it detects, Polgen creates new SE Linux types and generates policy rules. Because the dynamic behavior is insufficient to determine security policy, Polgen presents a wizard-style interface for human interaction. We call the interaction "guided automatic policy generation."

We designed Polgen primarily for security administrators who confront unfamiliar programs and are obliged to integrate them into existing policy. This paper highlights changes made to Polgen to adapt it to the needs of application authors, people that are less likely to be well versed in SE Linux policy than are security administrators. Key changes include an architecture specification language and a refinement of the wizard-style interface for application authors. When complete, this tool will expand the community of policy authors, and further accelerate the adoption of SE Linux.

1 Introduction

Security-Enhanced Linux adds mandatory access control to an open source operating system. Systems calls that fail to conform to a policy are aborted. Access control decisions are based on labeling system objects with types. To achieve realistic security goals, typical policies specify many types.

Because of the fine-grained use of types, it is difficult to extend a policy to support the security goals for a new application. Most extensions are written by SE Linux experts, with detailed knowledge of the existing policy infrastructure. Such extensions are needed to support new installations and new applications. Currently, the number of extensions is limited by the number of experts.

To support new installations, the Polgen guided policy generation tool was developed. It was initially targeted at security administrators, to aid them as they added new applications into their computing environment. It particularly focused on generating policies that allow the least set of privileges for items in an application, and assumed that the security administrator does not always have access to source code. In this mode of operation, Polgen uses dynamically collected information about system calls made by a program. From this information Polgen infers an information-flow graph. It then suggests policy modeled on patterns detected in that graph.

With so much of its original functionality focused on learning what a program does, we did not initially consider the utility-to-effort trade off for supporting policy generation by the actual authors of programs. However, Polgen contains an intermediate representation and conducts analysis on that representation in a way that is enabling us to capture program knowledge without requiring developers to deal with the syntax, and to some degree the semantics, of Type Enforcement (TE) policy. In addition, as we have progressed, we have constructed an interactive *guided automation* approach that we and others can extend to make policy generation available beyond the immediate SE Linux community.

Our current focus is to turn these powerful tools—pattern detection, information flow modeling, and guided automation—to better serve program authors. These authors may be assumed to have detailed knowledge of the operation of their programs, but only minimal understanding of SE Linux. Though they may be familiar with general security principles, they may have little knowledge of the specifics of SE Linux `.te` files. They probably can't tell `user_t` from `unconfined_t` or remember whether a file or a process usually gets the `exec_t` suffix. Our working hypothesis is that a combination of pattern modeling with guided automation is

*This work funded by NSA project 0705N6BZ-LS

perfect for encapsulating the knowledge of SE Linux that a program author needs to make a good first-cut policy.

Policy so generated may not be complete for all situations and it will not support the creation of elaborate modules with toggles to switch between vendor-specific and reference policies; however, our goal is to make it practical for program authors to produce policy modules no worse than they might now accomplish after substantial training and with significant attention to detail.

Section 2 describes the current Polgen system. In Section 3 we describe related work. Section 4 describes our current work including speculation on other forms of input we are considering for seeding the policy generation process.

2 Polgen

Polgen [4] is MITRE's tool for human-guided automated policy generation. It processes information flow data for a target program. By observing patterns [7] such as Pipeline, Interpreter, or Proxy in those flows, it generates policy appropriate to such patterns. Because the resultant policy makes a number of guesses to fill in absent information, Polgen presents a wizard-style interface for human interaction. We call this "guided automatic policy generation." The process is automatic, but admits human guidance and tuning.

2.1 Information flow capture

We build the information flow graph not only from dynamic `strace`s, but also from author generated models. These approaches generate complementary information. Currently, a user selects one or the other form. While some information merging is easily accomplished, inconsistencies and missing data can make merging difficult. We discuss this issue and our plans in the upcoming future work section. The next two sections cover our dynamic analysis and model-driven approaches, respectively.

2.1.1 `strace` and the Tracker

Polgen uses `strace` to identify dynamic behavior. To collect its traces of program behavior, the Polgen suite begins with a modified `strace`. Our patch to `strace` displays the security context of the executing process and of all resources accessed. It is a simple matter to execute a program under `strace` and exercise all desired behaviors. Unfortunately, full exercise of reasonable example programs (e.g., the Mozilla suite) produce enormous `strace` logs, often exceeding the standard file size limit.

```
jabberd = {
    reads_types {shlib, lib,
           proc, net_conf}
    listens_at {5222, 5347}
    connect_at {53}
    execs_types {bin} }

component router {
    parent jabberd
    reads_types {shlib, local_config}
    writes_types {local_log}
    socket_write {resolver, c2s}
    socket_read {resolver, c2s} }

component resolver {
    parent jabberd
    reads_types {shlib, local_config}
    writes_types {local_log}
    socket_write {router}
    socket_read {router} }

component c2s {
    parent jabberd
    reads_types {shlib, local_config}
    writes_types {local_log}
    socket_write {router}
    socket_read {router} }
```

Figure 1: Polgen Specification Language (PSL)

To pare the `strace` output down to a manageable size, we have written a program called the file-descriptor tracker (executable `tracker`). This consumes `strace` input through a pipe, and produces a succinct description of which security contexts are opened, read from, written to, or executed. The `tracker` is useful in its own right for producing type-focused descriptions of program activity. Recent work has reduced the `tracker`'s dependence on `strace` in favor of integration with the kernel audit system.

2.1.2 Modeling

Figure 1 shows an abridged description of Jabberd [14] as expressed in Polgen's specification language—PSL. PSL's heritage is the work on software architecture description languages [11]. It emphasizes description of components and connectors. Each component is described with the connectors it makes to the outside environment. Connectors include reads-from, writes-to, execs, and sockets. If the author knows the SE Linux type of an external component, he can enter that information directly. If not, he can simply enter a directory name and

Polgen will try to match it to the most likely type. In the figure, we see the description of four components—jabberd itself, and three of its sub-components. The router component has jabberd as a parent. It reads from the shlib directory and also a configuration file associated with jabberd. It writes to a jabberd log file component.

Descriptions in this language are a source for generating policy. We compile these descriptions into the same graph formalism that we use for dynamic traces. We also plan for eventual integration with other formats such as what is produced in Tresys work (see Section 3.1.2). In this way we can take advantage of the consistency rules used in an integrated development environment.

2.2 Pattern recovery

Having produced a list of interactions between security contexts using one of the two methods mentioned above, the Polgen suite analyzes these to produce semantically meaningful chunks, detecting and repairing violations of good security practice. A pair of tools are used: `typegen` marks elements of the new program, and `spar` recognizes patterns involving the interaction of those marked types.

`spar` operates in two phases. It first constructs an information-flow graph. It then walks this graph to search for patterns in the interactions of system calls. Patterns are small collections of typical interactions. Identification of instances of these patterns is useful in three ways.

First, a pattern can be used to identify meaningful subprograms. For example, `spar` will identify CGI scripts running in conjunction with a web-server. The value of this recognition is that the policy author can safely provide access to types of resources required by the subprogram without unnecessarily extending such privileges to the larger program and its other processes.

Second, a pattern may suggest the employment of special types to ensure that pattern constraints are enforced. For example, a pipe-and-filter pattern contains a start process, one or more intermediate processes, and an end process. Resources (e.g., files, pipes, sockets) that are used for information flow between processes must be restricted to modification only by the preceding process in the chain. This can be enforced by declaring these resources to be of a specific type, hence preventing arbitrary external processes from modifying their contents. `spar` will automatically change any of these resources that are from the new application (i.e., marked by `typegen`) to a new type.

Third, a pattern may entail type transition operations that are in conflict with other transitions. By capturing all patterns such as execution of a process, we can examine the type transitions for conflicts. `spar` has a strategy for resolving some of these conflicts. For example suppose we have a situation where our model suggests multiple processes of type A each read from a file of type B and execute a set of different processes of distinct types. But since we can only have one transition out of the (A, B) pair, `spar` will attempt to resolve the conflict. It offers various actions including merging a collection of types if they are all marked by `typegen`. More details will be provided later on this strategy.

2.3 Generation

2.3.1 Basic Approach

After completing its automated analysis of tracked files, `spar` provides a policy author with an opportunity to fine tune types prior to automated policy generation. `spar` allows users to modify type assignment strategies in several ways. Users can declare that `spar` should minimize the number of new types created or that `spar` should never change a type of any pre-existing types. If these options are not selected users will be able to manually decline creation of each new type as it is proposed. Finally, the user can specify a distinguished generic type, to support distribution specific differences between `unconfined_t`, `bin_t`, `user_t`, et al.

As another support for type assignment, `spar` allows users to set up ad hoc groups of processes. The notion is that you may want to establish new types for all files read or modified by processes in a group.

`spar` then allows users to confirm its type assignments. The user is able to select a group and pattern designation for a type or return to an original type. The default is to apply pattern and group constraints for `typegen`-created types and to use an original type for others. If you override one of these original types, `spar` will ask for confirmation. `spar` cleans up the remaining new-application types, and replaces them with a program-specific name. The final step is to set the directory level for all files with a new type. This information shows up in the `.fc` file and conforms to the regular expression protocols for `.fc` files.

Once the policy author has finished making changes, Polgen generates `.te` and `.fc` files. These files contain familiar "allow" and "type" statements. In addition, Polgen generates pattern instance macros. These macros are parameterized by the types of the processes, files and other resources that participate in pattern instances.

2.3.2 Reference Policy

Automatically generating reference policy requires tools to distinguish between resources of the program and external resources. For example, in Figure 1, local-config and local-log are key words that indicate a process will

respectively read from or write to a file of a new, private type. Such actions are "allowed" as in the basic approach. Other resource types such as `shlib_t` are external. Polgen calls out any actions concerning external resources in a separate interface file and display screen.

2.3.3 New types

Among the challenge areas for (semi)automated policy generation is the creation of new types.

Expanding existing types The primary challenge for any policy generation tool is the creation of new types. For example, the most pernicious failure of `audit2allow` is not that it permits everything—this might be controlled through careful review. It is that it never creates new types. We now present our approach to creating new types on a system with existing policy.

In order to manage the inheritance issue while creating new types and new relations between them, we have to show the user how each action changes the definition of types already on the system. We don't worry as much when he changes types local to this new module. But when he says that his new `prog_t` can write to `shadow_t`, this is an important change to the definition of `shadow_t`.

The basic operation of type creation is the *split*. The basic system-wide types such as `bin_t` and `user_t` already exist. When adding some new program, we can treat its executables as almost like `bin_t` and any users necessary to run it as almost like `user_t`. That is, we express the differences between the base types and these new types, then close the new types. In this way, we avoid the danger of inheritance in the actual `.te` files.

Minimizing type count One way to solve the problems above would be by providing new types for every file on a system, and for every process which might arise from the product of pre-existing processes and files. This way madness lies. Our goal is not only good security. We also have to produce a system that runs. We therefore devote some effort to reducing the number of types on the system. Types that are similar are presented to the user, with the suggestion that he might like to merge them. A *merge* operation is exactly the reverse of a split.

Grouping like types Humans generating policy know when to assign types to individual files, and when to assign types to directories and only explicitly label exceptional files. Such reasons are erased by the time an automated tool has the opportunity to act. It's hoped that package data will show us which files from a new application are being added to existing directories, and which are placed in entirely new directories. From such data, we can infer or guess when a change in many files in a directory should cause us to instead change the type of the directory.

3 Related Work

3.1 Other generation tools

We have developed Polgen in the footsteps of other generation tools—including the infamous `audit2allow`. We have also developed contemporaneously with several other projects providing advanced policy development support:

3.1.1 Virgil

Virgil [10] is a GUI tool for specifying simple SE Linux policies. It focuses attention on a single domain for easy use, but sacrifices the ability to describe interaction between domains. It has an easy interface for selecting capabilities without needing to exercise them, and a novel expression of "Subdomains" for scoped type transitions.

Unfortunately, Virgil must be used as a live GUI on a machine running the real policy, in parallel with the application for which policy is being generated, and with extensive privileges! It gains some critical information, such as the type for network sockets, from dynamic observation of the program.

Polgen provides a higher abstraction level for describing policies. It doesn't allow direct access to the Linux capabilities system, but also doesn't require knowledge of that system. Virgil and Polgen target different audiences: Virgil is for someone experienced with Linux and SE Linux, looking to quickly construct a simple policy. Polgen is targeted at those with less Linux familiarity, and provides support for more complicated policies, such as those with multiple domains.

3.1.2 Tresys IDE

Tresys has developed an Eclipse Plugin [3, 5]. As of the beginning of 2006, it has not been publicly distributed. It provides symbol completion, syntax highlighting, and some assistance for tracking the scopes of names. Unpublished discussions have indicated that it will support an annotated syntax for describing policies. This will enable interesting tools for merging policies or measuring the differences between them.

Such an export tool from the Tresys IDE would allow integration with Polgen. It might even be possible to integrate Polgen's pattern-detection and policy-suggestion components into the IDE for live suggestion of policy additions.

3.2 SELinux Policy Editor

Yuichi Nakamura, of Hitachi and the George Washington University, has developed a vastly simplified policy infrastructure [12]. By removing most of the file-type abstraction layer, he removes much of the difficulty and much of the expressive power of Type Enforcement. In particular, this system revives the inheritance problem of old hierarchical MAC systems that Type Enforcement was created to address.

This work addresses a distinct audience from Virgil and from Polgen: those with some experience in Linux, little to none in SE Linux, and with some familiarity with their application domain.

3.3 The macro language

The M4 macro language [15] has severe shortcomings as the primary abstraction of relations between types. It is excellent for aggregating and naming collections of `.te` rules, but this is all it can do. We would like to be able to map the patterns described above to macros. For example, when a pattern "Executable" is discovered, relating the type of some files `f_t` to the type of the processes that are run from them `p_t`, we would like to write a single line of `.te` file output `pattern_executable(f_t,p_t)`. What can we write in the definition of such a macro? It's easy to expand it to a call to the `domain_auto_trans` macro, but this is insufficient.

Other policy components may later wish to refer to the file from which `p_t` originated, or to the process which might erupt from `f_t`. Because these macros do not establish persistent data structures representing relations, and do not return values, we cannot easily write policy which is parameterized in this way.

Some policy manipulation tools [2] have performed limited versions of such relational maps. For example, if all policy authors obey the restriction that `can_exec(user_t,foo_exec_t,foo_t)` for all groups of related types `foo`, one can apply the map by adding or removing the string `exec_`. This only works in very strict circumstances. Most importantly, it cannot be extended to handle multiple orthogonal relations in the system. If some type of group `foo` participates in an unparameterized executable relation and a pipeline relation with groups `bar` and `baz`, should we name it `foo_exec_pipe_bar_pipe_baz_t`? We would very quickly create a world of horribly verbose pseudo-Hungarian types.

Still further insufficiencies become apparent once we try to maintain a policy generated largely from macros. For example, we cannot close a type. We cannot make use of relations established by non-macro contents of `.te` files, or even by other macros. For all these reasons, we have to maintain a higher-level abstraction of the intended policy, then compile that to the macro language or to direct `.te` rules. The pattern language provides such a higher-level abstraction. For reasons of maintainability, we choose to leave macros evident in the output `.te` files wherever possible.

3.4 System modeling and specification

Gathering high level information about a software system has been a long term goal of the software engineering community. The work most closely related to our notion of modeling systems can be found in the literature on software architectures [8]. While we borrow a basic ontology from this literature, the goal is different. The architecture work focuses on analysis of abstract artifacts prior to system implementation, while we use similar abstracts to start the policy generation process.

3.5 Pattern recovery

There is a rich archive of literature on software design patterns [7] that we have draw from. Familiar examples of patterns include Factory, Visitor, Pipeline, Mediator, and Chain of Responsibility. Most of these entail structuring to achieve a balance between strong typing (in the software engineering sense) and run-time flexibility.

The notion of design pattern is intentionally informal—the software community defines a pattern whenever variants are used over and over again to solve some problem. In some cases, two patterns will be structurally and behaviorally similar—the difference is in programmer intent. Thus, pattern recognition cannot be exact. However, it is possible to develop recognition assets that within reasonable error bounds will find known patterns in extant applications. Most of the work in software design pattern recovery has used static analysis techniques. Our own work [1] in developing a prototype tool called Osprey falls into this category.

Recently, there has been progress in combining static and dynamic analysis approaches for this task. Typically a static analysis will yield an abstract syntax tree. Dynamic analysis based on some form of instrumentation or debug support is then used to refine the analysis [9, 6, 13]. In contrast to that work, we are interested in the interaction among resources (processes, files, sockets, etc.), not classes and methods of an object-oriented program itself.

4 Future Work

There are several other sources of input to seed the Polgen process. In particular, if we have access to the pro-

gram's developers, we have excellent sources of information: direct interaction and existing data. We want experiences to get as much as possible out of existing data sources. This minimizes the burden on the policy author.

4.1 Guided interview

We want to keep the interview as short as possible while providing the most useful opportunities for the application author to describe the behavior of his program. The best compromise we can find is composed of these elements:

- Separation of questions which need answers from those which have reasonable defaults.
- Capture of interaction sequences for later playback. This prevents needless re-entry of data with program updates.
- Adaptation of the questions to an application author's mindset, rather than an SE Linux expert's mindset.

In some cases, we expect to use the language described above to provide a middle ground between an interview and direct `.te` files.

4.2 Light-weight static analysis

Analysis of distribution packages only produces information about the highest level of program components, and only about static components—files, not processes. It does not produce information about relations between processes from the new application and files already on the system.

Polgen now obtains such information from analysis of dynamic traces of the new program; however, dynamic analysis only works when an analyst exercises the relevant pieces of the code. Dynamic analysis can also consume a great deal of resources producing information already available to the program author.

The pattern recognizer used in `spar` descends from previous MITRE work on `Osprey`, a static-analysis based reverse engineering tool that recognizes software design patterns in C, C++, and Java code. Since the internal structures of `Osprey` and `spar` are quite similar (recognizers that match flow graphs to predefined patterns and constraints), we anticipate a smooth integration of these two approaches. When source code is available, Osprey can identify resources used by the program without the problems of dynamic analysis—and without requiring the author to re-specify these details.

To date, we have conducted static analysis on a few programs in order to confirm our dynamic analysis findings.

4.3 Integration

We have not yet dealt with the interplay between models, static analysis, and dynamic analysis. Indeed, the source code for some applications may not be available for use in static analysis. Where more than one mode is available we want to support policy authoring from multiple sources. We intend to look more closely at the work that generates an abstract syntax tree and then annotates this tree with information generated from abstract models and/or dynamic traces.

5 Conclusion

In the paper we have described Polgen, a prototype tool that program authors can use to generate SE Linux policy. These authors can provide a distinctly rich source of knowledge about their applications. Borrowing from research into software architecture and pattern descriptions allows us to capture that knowledge in PSL. The Polgen tool digests descriptions in PSL, together with optional dynamic traces, into information flow graphs. Our prior work on semi-automated generation of policy from such information flow graphs applies here. We translate the author's knowledge into information flow graphs, and the graphs into SE Linux policy.

We also explained our current plans to include guided interviews and light-weight static analysis to complement other forms of application information capture.

By augmenting the information flows from the pattern language with that from dynamic traces, we allow program authors to produce policy modules with much less effort and much higher quality.

References

[1] A. Asencio, S. Cardman, D. Harris, and E. Laderman. Relating expectations to automatically recovered design patterns. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 87, Washington, DC, USA, 2002. IEEE Computer Society.

[2] Tresys Corp. Setools policy tools for selinux, 2002–2005. URL `http://tresys.com/selinux/selinux_policy_tools.shtml`.

[3] Tresys Corp. Selinux development and integration ide, 2005. URL `http://www.tresys.com/selinux/sedev.shtml`.

[4] The MITRE Corporation. Polgen: Guided automated policy development. URL `http://www.mitre.org/tech/selinux/`.

[5] Eclipse.Org. The eclipse platform. URL http://www.eclipse.org/.

[6] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, NY, 2000.

[9] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Daniel H. Jones. Virgil: Selinux policy generator, 2005. URL http://sepolicy-virgil.sourceforge.net/.

[11] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997.

[12] Yuichi Nakamura. Simplifying policy management with selinux policy editor. In *SELinux Symposium 2005*, Baltimore, MD, USA, 2005. SELinux Symposium, LLC. URL http://www.selinux-symposium.org/2005/presentations/session4/4-2-nakamura.pdf.

[13] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, New York, NY, USA, 2002. ACM Press.

[14] Rob Norris, Justin Kirby, Stephen Marquad, and Jabberd Project. Jabberd 2 server. URL http://jabberd.jabberstudio.org/2/.

[15] Gary V. Vaughan and The GNU Project. Gnu m4 macro processor. URL http://www.gnu.org/software/m4/.