PoliSeer: A Tool for Managing Complex Security Policies

Daniel Lomsak and Jay Ligatti

University of South Florida
Department of Computer Science and Engineering
{dlomsak,ligatti}@cse.usf.edu

Abstract. Few tools exist for decomposing complex security policies into simpler modules. The policy-engineering tools that do exist either encapsulate entire policies as atomic, indecomposable modules or allow fine-grained modularization but are complicated and lack policy-visualization capabilities. This paper briefly presents PoliSeer, the first tool we are aware of that allows complex policies to be specified, visualized, modified, and enforced as compositions of simpler policy modules.

Key words: Security policies, policy engineering, policy visualization

1 Introduction

Although complex security policies are difficult to specify, analyze, and update, they arise often in practice. For example, a system administrator or end user may wish to enforce a complex collection of security constraints on an untrusted application to limit its access to resources such as files, memory, and peripheral devices and to obligate it to audit security-relevant operations and employ appropriate cryptographic protocols and intrusion detection on network communications. In general, software-security policies tend to become more and more complex over time, due to the emergence of new attacks, users' demands for relaxations to overly tight policy constraints, and the development of new application areas, like medical databases, which require domain-specific security and privacy considerations [2].

1.1 Related Work

The trend of increasing complexity in software-security policies mirrors the trend of increasing complexity in general software applications; however, many tools and techniques exist to help software engineers specify, analyze, and update complex software applications. One of the most common techniques is modularization; engineers can modularize software into independent, reusable, individually testable components (packages, classes, functions, aspects, etc.) that can be parameterized by, and can communicate with, other components through well-defined interfaces. Decomposing complex software into simpler modules saves engineers from having to manage software as a single, indecomposable code block.

Integrated development environments (IDEs) for software engineering typically provide good support for navigating and visualizing software modules [7, 14, 15].

In contrast, we are not aware of any existing policy-engineering tools that enable users to navigate and visualize arbitrary policy modules, or even to build arbitrary policies straightforwardly as compositions of existing policy modules. Many policy-engineering tools do exist for specifying and managing—without visualizing—arbitrary but uncomposable policies (e.g., [6, 8, 9]). Many other tools exist for specifying, visualizing, analyzing, and/or composing policies, but only in particular domains; e.g., Policy Visualization Analysis tool provides a GUI for managing SE Linux policies [16], Expandable Grid manages access-control policies [13], Policy Mapper manages location-based access-control policies [4], front ends for SPARCLE and PERMIS manage natural-language access-control and privacy policies [5, 10], and Fang and Firmato manage firewall policies [12, 1]. None of these tools allow policy engineers to specify, manipulate, compose, visualize, or enforce arbitrary (including cross-domain) policies.

Another related tool is Polymer, which does allow cross-domain policies to be specified and enforced as compositions of simpler policy modules [2]. Polymer users specify runtime policies on Java-bytecode applications in the Polymer programming language. Polymer policies exhibit *universal composability*; every policy can be composed arbitrarily with other policies. Polymer achieves universal composability by (1) making all policies first-class objects (i.e., objects that are treated like all other values, which can be passed as arguments to and returned as results from methods) and (2) requiring all policy objects to implement a standard interface. A Polymer policy P can be parameterized by another policy P', and when P has to decide whether and how to allow a security-relevant application event A to occur, P may query P' for a response to A and use that response to generate its own response. For example, a `Conjunction` policy might be parameterized by two policies P_1 and P_2; the overall policy can enforce the conjunction of P_1 and P_2 by always responding to security-relevant events with the most restrictive of the responses of P_1 and P_2. In this case `Conjunction` is a *superpolicy*, and P_1 and P_2 are *subpolicies*. As another example, an `Audit` superpolicy may be parameterized by a policy P and a string S; then `Audit` can blindly enforce P while logging all security-relevant events to file S. In this way, policy engineers can build arbitrary, complex policies as compositions of simpler subpolicies.

However, Polymer lacks policy-visualization capabilities, and policy engineers cannot specify, analyze, or update policies straightforwardly in Polymer due to its complicated static and dynamic semantics. The complications stem from Polymer's safeguards to ensure universal composability; policies must segregate effects (observable state updates and I/O operations) out of `query` methods and into `accept` and `result` methods. Segregating effects makes for complicated control flow between policy methods, and policy engineers must understand this control flow in order to specify, analyze, and update policies correctly. For example, Figure 1 contains an email-client policy with `query`, `accept`, and `result` methods, as specified in Polymer, while Figure 6 shows how the same policy could be straightforwardly specified and visualized in PoliSeer.

```
package examples.mail;
import polymer.*;
import examples.*;
public class Email extends Policy {
  private Policy p;
  public Email(){
    p = new Audit(new Conjunction(new Dominates(new Dominates(new Reflection(),
          new Conjunction(new ClassLoaders(), new NoOpenClassFiles())),
          new Dominates(new DisSysCalls(), new InterruptToCheckMem(10.0, 4000))),
          new IsClientSigned(new Trivial(), new Dominates(new Dominates(new TryWith(
          new ConfirmAndAllowOnlyHTTP(), new AllowOnlyMIME()), new Attachments()),
          new Conjunction(new OutgoingMail("PoliSeer@cse.usf.edu"),
          new AllowInsertedActions(new IncomingMail())))))), "email.log");
  }
  public Suggestion query(Action a) { return p.query(a); }
  public void accept(Suggestion s) { p.accept(s); }
  public void handleResult(Suggestion s, Object result, boolean wasExnThn) {
    p.handleResult(s, result, wasExnThn);
} }
```

Fig. 1. Email-client policy specified in Polymer (taken from [3]). The same policy can be specified and visualized in PoliSeer as shown in Figure 6.

1.2 Contributions

We briefly present PoliSeer, the first tool we know of that enables complex policies to be straightforwardly specified, visualized, modified, and enforced as compositions of simpler subpolicy modules. PoliSeer users can import universally composable policies from a policy library, compose them in meaningful ways by declaring arguments to policies, visualize and update policies holistically, and generate code to enforce the composed policies. Our prototype implementation of PoliSeer uses Polymer as an underlying language of universally composable policies (i.e., our PoliSeer implementation imports and exports Polymer policies); however, we have designed PoliSeer to be readily portable to any other system with universally composable (first-class and parameterized) policies.

Interested readers will find a much more detailed presentation of PoliSeer in our companion technical report [11].

2 The PoliSeer Interface

PoliSeer is a simple IDE for security policies. It provides a straightforward interface for creating, visualizing, modifying, and enforcing complex policies.

2.1 The Main Window

The main PoliSeer window consists of two panels, as shown in Figure 2. The left is the *policy-selector panel*; the right is the *policy-tree panel.*

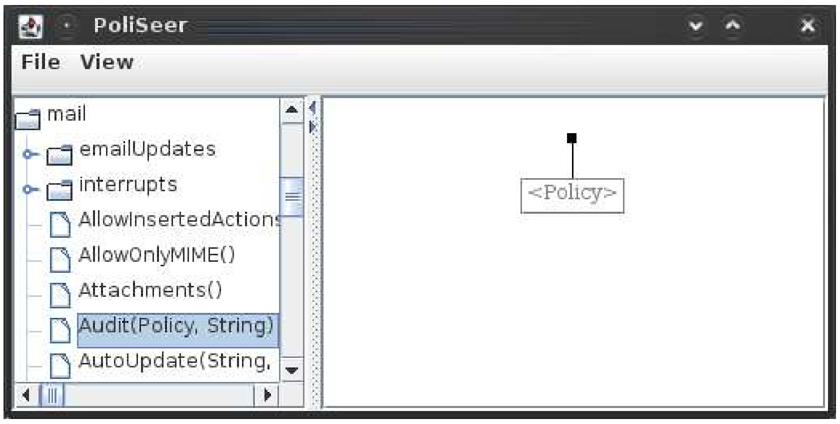

Fig. 2. Main PoliSeer window divided into policy-selector and policy-tree panels. The policy-tree panel is displaying the default, empty policy.

– The policy-selector panel allows users to navigate the file system to find existing composable policies. Our implementation begins by populating the policy-selector panel with all subdirectories and `.poly` files (i.e., Polymer policy files) in the user's home directory. The policy-selector panel uses a standard interface for navigating the file system: clickable areas expand and contract subdirectories. When a user expands a subdirectory, PoliSeer searches for, parses, and displays all policy files in the newly visible directory. PoliSeer parses the policy files so it can display the types of parameters each policy expects (in its constructor) next to that policy's name in the policy-selector panel, as shown in Figure 2.
– The policy-tree panel contains a visualization of the current policy. When PoliSeer begins executing, it displays the empty policy as shown in Figure 2. The empty policy consists of a single grayed-out node containing the text `<Policy>`, which indicates that PoliSeer expects that node to be filled in with a policy. In general, grayed-out nodes in a PoliSeer policy indicate incompletions in the policy; the text in a grayed-out node indicates the type of data that must be inserted into that node. In this way, PoliSeer communicates to the user whether, and in what ways, policies are incomplete. For example, Figure 3 shows a policy-tree panel for an incomplete, one-node `Audit` policy parameterized by another `Policy` and a `String`; the policy is incomplete until the user specifies a `Policy` and a `String` argument for `Audit`.

2.2 Creating Policies

PoliSeer's basic interface for creating policies is simple. Users may select a policy in the policy-selector panel by left-clicking on the policy name. Having selected a

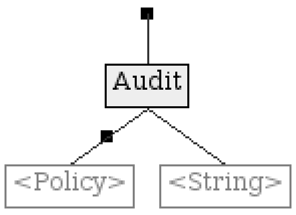

Fig. 3. Policy-tree panel showing a root `Audit` policy parameterized by another `Policy` and a `String`, though no children have yet been specified.

policy P, the user may left-click on any *landing area* L in the policy-tree panel to insert P into L. Valid landing areas are grayed-out `<Policy>` nodes and *branch-insertion points* (BIPs) in the policy-tree panel. PoliSeer automatically displays BIPs as small black squares in the policy-tree panel on every branch into which a user could possibly insert a policy.

For example, Figure 2 shows PoliSeer as it begins, with an empty policy-tree panel. A user may add the `Audit` policy as the root of the policy tree by clicking on the `Audit` policy in the policy-selector panel and then clicking on the grayed-out `Policy` node in the policy-tree panel. The policy tree in Figure 3 results from this addition; `Audit` has been added as the root node of the policy, but two new grayed-out nodes have appeared because PoliSeer has parsed the `Audit` policy and determined that it is parameterized by another `Policy` and a `String`. The user may then insert a `String` as the right child of the `Audit` policy by clicking on the grayed-out `String` node and entering the string in a pop-up window. Figure 4 shows a complete policy tree that results from inserting a (childless) policy and a string into the grayed-out nodes of Figure 3. Two BIPs exist in Figure 4; a user may insert a policy node into this policy above the `Audit` root or above the `DisSysCalls` child of `Audit`. To insert a `Conjunction` policy between the `Audit` and `DisSysCalls` nodes, the user simply clicks on the `Conjunction` policy in the policy-selector panel and then clicks on the BIP between the `Audit` and `DisSysCalls` policies in Figure 4; the result is shown in Figure 5.

Having created a (complete or incomplete) PoliSeer policy, a user may save it to a `.psr` file (which is simply a serialization of the policy tree) with the `File -> Save Tree` menu option and may generate Polymer code to enforce the policy in a `.poly` file with the `File -> Generate Policy Code` option. Conversely, users may resume creating, viewing, or modifying a saved `.psr` policy with the `File -> Load Tree` option. When exporting an incomplete PoliSeer policy to a `.poly` file, PoliSeer automatically parameterizes the exported policy by all missing policy components (e.g., if the PoliSeer policy is complete except for missing one child of a `Conjunction` superpolicy, then the exported policy will have a single parameter, a policy that fills in for the missing child of `Conjunction`).

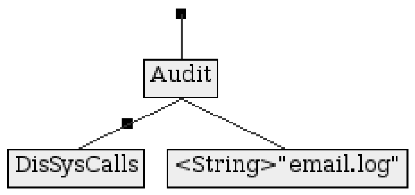

Fig. 4. Policy-tree panel showing an `Audit` policy with subpolicy and string arguments. This complete policy disallows system calls (i.e., `java.lang.Runtime.exec` methods) at runtime while logging all policy decisions to a file named `email.log`.

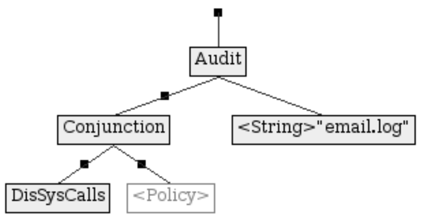

Fig. 5. The same policy-tree panel shown in Figure 4, except that the user has now inserted a `Conjunction` policy between the `DisSysCalls` and `Audit` policies.

2.3 Visualizing Policies

As Figure 6 demonstrates, PoliSeer's policy-tree panel can provide a useful high-level visualization of complex policies decomposed into subpolicy modules. If PoliSeer's view of a policy tree is too high level, users can always choose the `View -> Policy Source` menu option to obtain the source-code-level details of the most recently selected policy.

2.4 Modifying Policies

PoliSeer users may also modify existing policy trees; please see our technical report for details [11].

3 Conclusions

PoliSeer is the first tool we know of to allow policy engineers to specify, visualize, modify, and enforce complex policies as arbitrary compositions of simpler subpolicies. As described in our companion technical report [11], we have implemented PoliSeer using the architecture in Figure 7, have measured the performance of our implementation, and have designed and enforced an extensive case-study policy with PoliSeer.

We view PoliSeer as a simple IDE for security policies, providing policy engineers the same sorts of benefits that traditional IDEs provide software engineers (convenience of creating high-level specifications and visualizations to minimize errors in, or totally avoid, low-level programming tasks). We hope that with continued research and development, policy IDEs will be as helpful for managing complex security constraints as standard IDEs have become for managing complex software.

Acknowledgments. This research was funded in part by National Science Foundation grants CNS-0742736 and CNS-0716343.

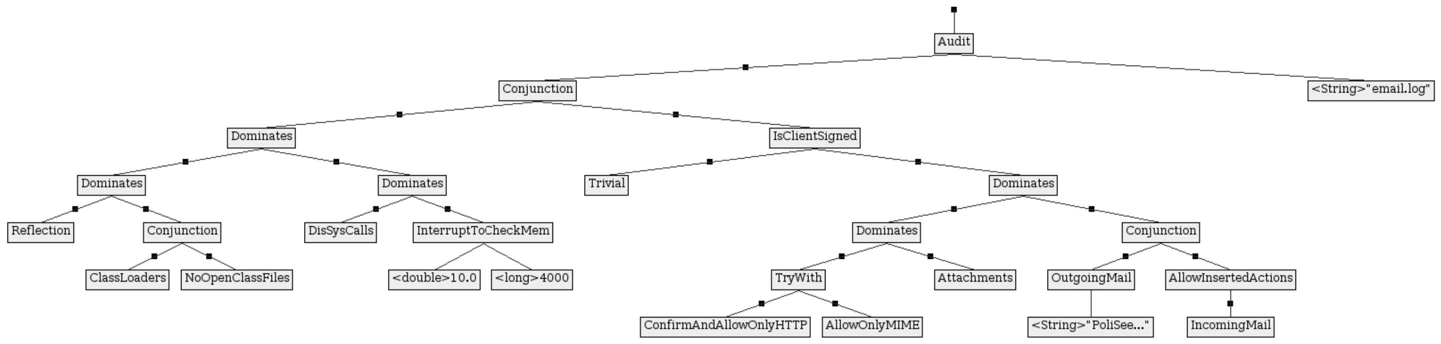

Fig. 6. Full tree for an example email policy (taken from [3]).

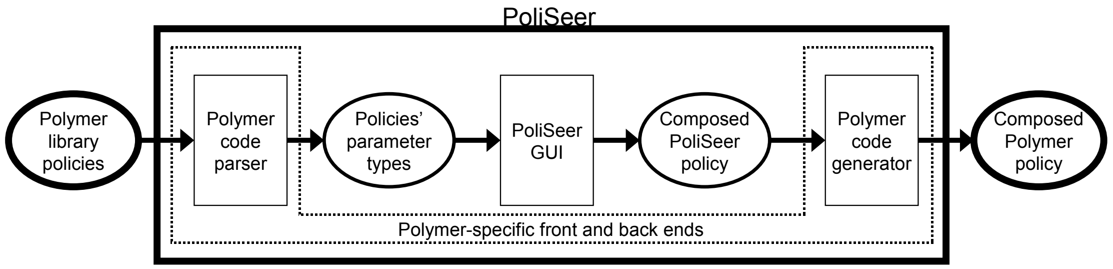

Fig. 7. Architectural overview of PoliSeer implementation (described in our companion technical report [11]).

References

1. Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.
2. L. Bauer, J. Ligatti, and D. Walker. Composing expressive run-time security policies. *ACM Transactions on Software Engineering and Methodology*, 18(3):1–43, 2009.
3. L. Bauer, J. Ligatti, and D. Walker. Polymer: A language for composing run-time security policies, 2008. `http://www.cs.princeton.edu/sip/projects/polymer/`.
4. R. Bhatti, M. Damiani, D. Bettis, and E. Bertino. Policy mapper: Administering location-based access-control policies. *Internet Computing, IEEE*, 12(2):38–45, March-April 2008.
5. C. A. Brodie, C.-M. Karat, and J. Karat. An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench. In *Proceedings of the second symposium on Usable privacy and security*, pages 8–19, 2006.
6. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–39, 2001.
7. S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer-Verlag, Berlin, 2007.
8. D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, 1999.
9. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):158–173, 2004.
10. P. Inglesant, M. A. Sasse, D. Chadwick, and L. L. Shi. Expressions of expertness: the virtuous circle of natural language for access control policy specification. In *Proceedings of the 4th symposium on Usable privacy and security*, pages 77–88, 2008.
11. D. Lomsak and J. Ligatti. *PoliSeer: A Tool for Managing Complex Security Policies.* Technical Report CSE-SSec-112509, University of South Florida, 2009.
12. A. Mayer, A. Wool, and E. Ziskind. Fang: a firewall analysis engine. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–187, 2000.
13. R. W. Reeder, L. Bauer, L. Cranor, M. K. Reiter, K. Bacon, K. How, and H. Strong. Expandable grids for visualizing and authoring computer security policies. In *CHI 2008: Conference on Human Factors in Computing Systems*, pages 1473–1482, Apr. 2008.
14. N. Saigal and J. Ligatti. Inline Visualization of Concerns. In *Proceedings of the International Conference on Software Engineering Research, Management, and Applications (SERA)*, Dec. 2009.
15. T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE Transactions on Software Engineering*, 32(9):753–768, 2006.
16. W. Xu, M. Shehab, and G.-J. Ahn. Visualization based policy analysis: case study in selinux. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 165–174, New York, NY, USA, 2008. ACM.