Embracing Policy Engineering

Kathi Fisler
WPI
Worcester, MA, USA
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
Providence, RI, USA
sk@cs.brown.edu

Daniel J. Dougherty
WPI
Worcester, MA, USA
dd@cs.wpi.edu

ABSTRACT

Declarative policies play a central role in many modern software systems. Engineering policies and their interactions with programs raises many interesting open questions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Design, Human Factors, Languages, Security

Keywords

Policies, policy engineering, program-policy interactions

1. A POLICY PRIMER

Consider a hospital's electronic health-records system: doctors need to record treatments and to access medical history across multiple providers; nurses need the ability to record administered medications, but only while they are on duty in the patient's current hospital ward; the billing department needs to determine what treatments were ordered to prepare insurance claims, but don't need details of diagnoses. Rules such as these collectively form the application's access-control policy. Increasingly, software applications separate these rules from the main program logic, capture them though policy-specification languages, and consult them via an engine or reference monitor when users attempt to perform actions within the software system.

The health-records example hints at the complexity of realistic policies. Decisions are based on nuanced criteria (such as whether and where a nurse is on duty and the purpose of accesses); the simple "subject/action/resource" model of file permissons and classic access control lists is too weak for most applications. Users may hold credentials for multiple roles (such as a doctor who is admitted as a patient), resulting in inappropriate access to information. The association between people and their credentials changes frequently as staff go on- or off-duty and patients are admitted and discharged. Consultation between physicians requires temporary access to sensitive data. Different access rules apply to different information within the same person's record (most states, for example, protect mental health records and HIV status more carefully than other medical information). Policies must comply with regulatory requirements such as HIPAA. In light of these myriad situations, realistic policies can contain thousands of rules over hundreds of roles and resources, as well as huge databases mapping individual users to their roles and other credentials. Not surprisingly, policies are much like programs in their need for requirements, development processes, validation, and ongoing maintenance.

Software engineers should thus find policy development, validation, and maintenance familiar and yet different enough to be interesting. Policies are a key component of security and regulatory compliance, both of which are core software engineering concerns. Policies also offer interesting twists to established software engineering problems.

Modularity.

Given that policies are generally maintained in separate files from the main program logic, they engender a form of program modularity. This form is interesting because policy modules are written in declarative languages that are less expressive than the languages used for writing the overall application. Datalog, for example, is a common underlying foundation for policy languages; XACML is an example industry standard. From a software-engineering perspective, this modular structure is interesting for two main reasons:

- *Policies are amenable to different analysis techniques than their host applications.* Since policy languages are less expressive than application languages, policies are often amenable to richer analysis than are programs. For example, we can compute the difference between two policies, whereas computing the difference between two arbitrary programs is generally infeasible. This separation of expressive power also enables certain security analyses that would not be possible if the entire program, including policies, were written in the main application language. For example, we can determine that a policy will never grant a particular access as long as a certain configuration of program state is not reachable. The complex security analysis occurs on the simpler policy, leaving only a safety property to prove of the more complex program.

- *Policies and application code are maintained by different people.* Many policies are maintained by human resources, legal departments, or end-users of software systems (as in privacy settings in social networks). These users have very different needs in the details of policy engineering than the developers of the overall application, who are more technically trained. Different engineering processes are therefore required for different components within the overall system.

In practice, applications contain many policies governing different levels of abstraction. A web-based software application with a database back-end would have server-side policies on user actions, policies within the database, file permissions, and some firewall or network level policies. The "policy" of a real application is not a single file in a single policy language. This makes the modularity arising from policies all the more interesting.

Human Factors.

Access-control and privacy policies often capture subtle and sensitive human relationships. A friend request in a social network, for example, implicitly asks a user to consider how much information to trust the requestor with and the consequences of refusing the request. Users concerned both with maintaining privacy and appearing friendly end up managing subnetworks of friends or not utilizing the network. As friend settings are a form of access-policy authoring, this example illustrates the human issues that come to bear on policy authoring in some domains.

Similar concerns can arise in enterprise policy authoring as well. In our own work on policy authoring, for example, we found some users uneasy that settings based on least privilege would imply a lack of trust in co-workers. For public-facing applications, enterprises encode statements of values in the information they choose to disseminate. Many papers document the essential role of social concerns in privacy settings.[1] Thus, policies raise human-factors issues that fall outside conventional techniques focused on usability. The entire development process for policies must translate social concerns and relationships into the data and events embodied in an application.

Regulatory and Legal Issues.

Privacy and data protection are embodied in laws and regulatory controls, such as HIPAA (health information) and FERPA (student information). Several researchers are exploring how to extract and verify formal software requirements from such legislation. Automating compliance handling in software is complicated by differences in privacy laws across countries that may use the same software. Interesting legislative questions are also on the horizon. In the case of medical records, for example, questions remain about who owns the data (the patient or the hospital) and what controls individuals should have over personal data held on their behalf by organizations. Software systems that manage sensitive data will need to adapt to regulations as they evolve differently worldwide.

[1]e.g., L. Palen and P. Dourish, *Unpacking "Privacy" for a networked world*, SIGCHI 2003; T. Whalen, D. K. Smetters, and E. F. Churchill, *User experiences with sharing and access control*, CHI Extended Abstracts 2006.

2. OPEN PROBLEMS

The previous discussions hint at some of the interesting open problems in policy engineering. These include:

1. Tractable models and analysis techniques for the interactions between policies and the software systems that use them. Most policy-analysis papers ignore the surrounding software system, thus losing valuable information about which requests for access are even reachable and under what conditions.

2. Analysis techniques that help gauge whether a policy is overly restrictive for the workflow it manages. Studies show that people circumvent access controls and other security features that interfere with completing essential tasks. Are there ways to look for key workflows in software system models, and to determine whether policies are unnecessarily impeding any of them?

3. Analysis techniques that fit the low technical expertise of many policy authors. Property-free techniques—such as our work on change-impact analysis,[2] which computes all access requests whose decisions are different under two policies—help such users apply formal methods without needing to themselves be formalists.

4. Policy-maintenance techniques that help transfer trust in the current policy to a revised policy. Given the frequency of policy edits, policy managers should not have to analyze a policy from scratch on each edit. Change-impact analysis is one approach to this, but others surely exist as well.

5. Policy-engineering processes that work in small organizations or other lightweight settings. Most security-engineering processes start from assumptions about security goals and risk analyses that are unrealistic for small organizations. What processes might we develop to help these users think through the security requirements that would impact policies?

6. Migrating policies, such as security settings, across software architectures. As more security-sensitive applications shift to the cloud, policies will need to migrate to the cloud as well. How do the new abstractions underlying cloud architectures affect policies? Can we provide automated assistance to help developers update their policies during migration?

7. Developing flexible policy frameworks that can deploy the same logical policy to heterogeneous devices or systems with incompatible abstractions. At an industrial-experience panel at SACMAT 2010 (an ACM conference on access-control), a security-team leader for a Fortune-500 company stated that they have nearly 3000 active policies, many covering the same high-level requirements, but on different computing platforms or software versions. Differences in the APIs and abstractions in these tools means that the policies implementing the same requirements are often logically inconsistent by necessity. Can we create tools that help adapt policies to different system configurations?

[2]K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz, *Verification and change impact analysis of access-control policies*, ICSE 2005.