Practical Policy Patterns

Dan Thomsen
Independent Consultant
Minneapolis, MN
612 789-0559
d.j.thomsen@ieee.org

ABSTRACT

The paper attempts to encourage deeper thinking about the nature of security enforcement policies with the intent of fostering a practical engineering design approach for building security enforcement policy. The paper suggests several approaches to lower the cost of developing security enforcement policies by developing technology to share enforcement policies like open source software, including patterns, isolation of site specific policy and tools to increase the ability of humans to understand the implemented policy. The paper also suggests research avenues for increasing human understanding of enforcement policy.

Categories and Subject Descriptors
D.4.6 [**Operating Systems**]: Security and Protection – *Access controls*
D.2.4 [**Software Engineering**]: Software/Program Verification.

General Terms
Security.

Keywords
Security pattern engineering, computer security, access control.

1. INTRODUCTION

A security enforcement policy represents the configuration of an enterprise's security mechanisms. Creating a secure enforcement policy requires a deep understanding of the software involved and a deep understanding of the protection goals. A good least privilege policy requires a large amount of detailed knowledge on how users and applications interoperate and consume enterprise resources. The complexity of developing a good security enforcement policy approaches that of software engineering. While the implementation of an enforcement policy requires less effort than developing software, the software engineering mindset produces more effective and secure enforcement policies.

Most enforcement policies evolve from the reaction to day-to-day events, as opposed to a thoughtful design process. Functionality drives this day-to-day viewpoint more than security requirements. Effective security requires a mandatory policy that satisfies easy to understand properties. A discretionary policy leaves security in the hands of the users, who simply will not enforce a consistent policy even without the threat of Trojan horses.

As Figure 1 shows, this paper proposes to shift the burden of creating the security policy away from day-to-day administration tasks to a structured, bottom up, design process accomplished by security policy engineers. The approach applies engineering design patterns to creating security enforcement policies.

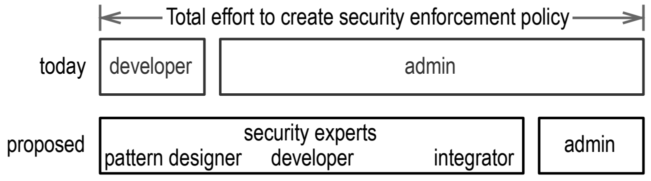


Figure 1. Shift effort from administrator to security experts.

Unlike software development, the simpler implementation of enforcement policies makes it possible to create an architecture that automates the translation from easily understood enforcement policy to site specific enforcement mechanism policies.

The paper discusses the challenges that have prevented a software engineering mentality in the past, and sketches a solution for each. Section 3 summarizes how all the elements fit together to create an architecture that reduces costs for creating enforcement policy.

2. CHALLENGES

There are four key challenges in making an engineering discipline for security enforcement policies;

1) Overcoming the incorrect mentality that creating security enforcement policy is easy.
2) Creating translation tools that spans the existing security models and enforcement mechanisms to allow designers to choose the best tool for the job.
3) Separating site specific details from generic patterns.
4) Understanding the impact of the resulting enforcement policy.

The paper discusses each challenge below.

2.1 Taking Enforcement Policy Seriously

Creating an accurate, secure enforcement mechanism that meets the security needs of an organization takes a great deal of effort. Often the enforcement policy evolves over time as the organization installs new capabilities, obscuring the full cost and resulting in a poorly understood, and insecure policy. Researchers often fare no better, often underestimating the effort to create a good enforcement policy for a 4new mechanism or security model.

Economics provides the most effective motivation for taking enforcement policy seriously. Organizations need a return on investment argument to motivate giving enforcement policy more attention. A good enforcement policy has reduced security incidents and limits the potential damage a malicious insider could do. However, it is often simply cheaper for organizations to cover the losses than invest in building an effective policy.

Reducing the cost of developing enforcement policy to nearly free becomes the only way to solve the problem. The open source software development model presents an effective "nearly" free approach that researchers could apply to creating enforcement policy. Currently much of the same policy engineering happens at every organization. Eliminating this duplicated effort spreads the development cost across many organizations, reducing the cost for a specific organization to provide a clear return on investment.

Just as a small business does not develop its own web server, it should also not develop its own security enforcement policy, if that is not its expertise. Many software vendors do provide policy elements for their product. However the diversity of security mechanisms and models means the local administrator must still understand each one. This tower of babble also obscures the total enterprise policy by distributing it through a variety of mechanisms. Strong security requires tools that manage this diversity to improve understanding.

2.2 Security Mechanism and Model Diversity

Security models and enforcement mechanisms range from permission bits on a file system to stored procedures in a database management system. Consider the Protection Proxy [5] pattern that runs arbitrary code to check some security condition. Once a security mechanism incorporates a proxy capability, all sound access control models and mechanisms become isomorphic. In other words you cannot express some policy in one mechanism that cannot be expressed by all other sound mechanisms. Clearly some security models make expressing some policy types easier, and some models require extensive proxies to express the policy. Diverse security models allow developers to select the model with the ideal cognitive fit for capturing key protection goals. Unfortunately, in practice often the security model fits the protection goals poorly, leading to human error. Creating an understanding that spans security models, allows policy developers to choose the best tool for the job.

All sound security models being isomorphic requires a formal proof. However, incorporating the word "sound" eliminates many fringe models that cannot implement effective security policy. Consider a primitive Unix file system with only nine permission bits. Such a model can implement an assured pipeline, which is a series of proxies connected together [1] but it cannot satisfy the requirement for the later stages to only read data from the prior stage because of globally readable files [13]. Models that cannot control the flow of information cannot implement serious security.

The isomorphic claim comes from practical experience creating a series of enterprise spanning policy specification tools that translate into existing enforcement mechanisms [14][15][8][12]. These tools allowed for specification of the policy in a single model and translated to a variety of mechanisms like; file permissions, firewall rules, authentication tools, and databases. The approaches represent a bottom up approach that group resources into a hierarchy of capabilities.

This yields an interesting question, "What is the simplest access control model that spans all sound security models?" What is the Turing machine of access control? The Lampson access matrix that maps all subject to all objects, certainly could express all policies [6], but does not aid humans in understanding the resulting enforcement policy. Engineers do not write flight controllers on Turing machines. Conceptual tools for proofs do not necessarily yield engineering tools.

A universal security policy represents an interesting academic question. However, from a practical engineering viewpoint the value comes from the ability to translate between security models to increase human understanding of the policy. Economics often dictate the enforcement mechanism, but with translation, designers can select the best model for the security requirements and simply translate it to the economically dictated mechanism.

This paper presents a simple variant of the NIST RBAC model [9] to aid understanding the benefits of translation. The model consists of actors and resources. Resources represent the finest granularity object the policy can reference. Assigning a resource to an actor implies the actor can access the resource. Resources allow designers to abstract out permissions and mechanism differences and work with simple sets of resources. The model organizes resources needed to perform a specific task into a set for easy assignment to actors who perform those tasks. Designers can choose to group sets together to conveniently capture common patterns. The set structure forms a hierarchy of membership, equivalent to an object-oriented design hierarchy for assigning resources to actors.

2.3 Separating Site Specific Details

When an application developer creates an application, they understand the tasks users can perform with it. The developer can also list the resources the application needs to accomplish these tasks. The developer can statically define some resources, like the password file, while other resources depend on local configurations, such as the path to user directories.

If applications defined the resources they need along with the necessary security properties, these classes could be expanded and mapped to the local environment. This requires more effort on the application developer's part, but it results in embedding security understanding directly in the development process. It also makes it possible to systematically separate out the common engineering patterns in access control.

Figure 2 shows the NIST core RBAC model for quick comparison [9]. Figure 3 show a slightly adjusted model that enables better site independence. Figure 3 changes *user* to *actor*, divides *role* into *role* and *task*, refines permission assignments into *resource interface*, *container*, and adds an *object class*. The role and task concepts are 100% equivalent, except application developers define tasks independent of any local configuration information. In essence, Tasks become the site independent roles in the role hierarchy.

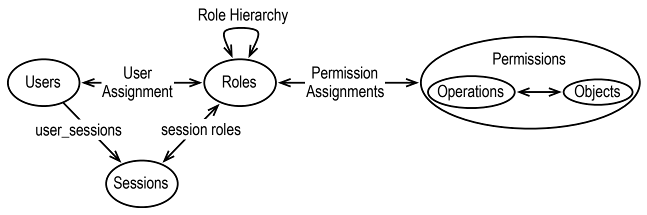


Figure 2. Standard NIST Core RBAC model.

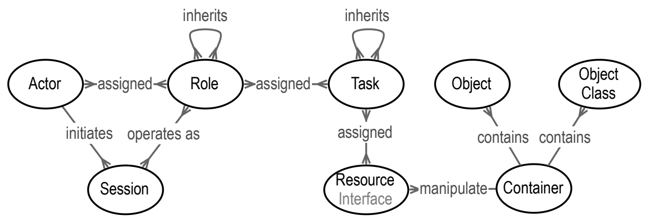


Figure 3. Modified NIST RBAC model.

Figure 3 replaces the user in the NIST model with actor to allow for more extensive definition of policy. Actors include users, programs, or hosts. Actors also include implied actors that define a set of actors based on some predicate, such as all users that have access to this network segment, or all actors in the enterprise. The actor definition allows the policy developers to define constraints independent of the actual users and configuration information.

Figure 3 also refines the NIST permission assignments. The approach defines objects and object classes. Object classes define a set of objects with the same security context pattern. Object classes allow the model to express a class of access control patterns. Different actors often access objects the same way. For example, suppose each user has a unique home directory, with specific permissions as the directory owner. Rather than tediously specifying the permissions for each home directory, the developer can define an object class that captures the pattern. The object class expands into a set of objects when translated to the enforcement system.

The model also introduces the *resource interface* concept. The resource interface defines the access to the object, such as read/write or a database view that selects some subset of rows and columns from a database table. The resource interface represents the proxy pattern that determines access for all objects in the container and could represent permissions or custom proxy code.

The *container* concept improves human understanding of the security policy when complex security constraints interact, which the next section explains further.

2.4 Adding "Why" to Security Enforcement

Enforcement policies have a life cycle that spans years just like software. Good software development includes documentation on the intent of the software model, but the intent of the original policy creator rarely gets documented.

Software exists to provide functionality. Functionality is the "why" of software and good regression testing can catch when a software modification destroys needed functionality. The "why" of enforcement policy exists outside the implementation and remains untestable. Standard regression testing can ensure that enforcement policy changes do not impact functionality, but it cannot ensure the change meets the original intent of the policy. For example, why do Alice and Bob have to communicate via an encrypted channel? Regression testing could show that the policy changes still allow access to the cryptographic keys and network, but cannot show Alice's sensitive data doesn't fall into the malicious Chuck's hands via some new channel.

Clark and Wilson formalized the need for adding the "how" to the "who" and "what" of access control to create the Clark-Wilson triple [2]. Understanding the policy requires adding a fourth element to the access control tuple, the "why". Without the "why" it becomes impossible to understand the intent of the policy.

A day-to-day evolutionary approach to enforcement policy rarely results in documenting the intent of the policy, simply due to the required extra investment of time. However, amortizing the cost across organizations through reusable patterns reduces the cost.

The container concept was added to the NIST model to create better human understanding of the resulting policy. Most systems have hundreds of thousands of objects, which policy designers cannot address individually. A container groups objects into sets. All the objects in the container are accessed the same way, through the resource interface. Each object belongs to one and only one container. Actors can access an object only through the defined resource interfaces. Containers cannot contain other containers, because it would violate the rule that an object belongs to only one container.

The container concept aids human understanding by clarifying to the user that the only way to access an object in the container is to go through the resource interface. For example, suppose a container contains a medial record object. A variety of user roles need access to the record depending on the current situation; for example, an emergency room doctor might only be allowed to access the record for patients currently in the emergency room. To support this constraint a resource interface must define an emergency physician interface and what constraints must be satisfied before allowing access. This allows for implementing a Team Based access control [11].

The interfaces to the container provide a list of all access to objects in the container, allowing a human to review and understand who can access objects at any time in the life cycle. For example, if the object contains a database table, interfaces may slice and dice what information actors can see. Without access to a container interface an actor cannot see anything in the container. Many different interfaces may in fact grant access to an element in a table under different conditions, but the interfaces represent a complete list of the conditions. Humans must still understand the interactions of these conditions to ensure the conditions meet the intended policy. Proper container design can limit the need for understanding complex conditional interactions.

Encoding the intent of a policy element requires further study. Other concepts may improve human understanding of the policy more effectively than containers. Once possible approach may capture what threat the interface was designed to protect against. While a text description clearly captures the intent, it may be possible to build a model of the threat the resource interface addresses. Then automated reasoning tools could ensure validity of the protection goals over the policies life cycle. Clearly a variety of constraints and security invariant clauses could be created, but often such constraints do not include the active nature of an adversary trying to bypass the security policy. For software, humans still out perform software tools for reasoning about intent. However, the simpler domain of enforcement policy makes automated reasoning more tractable.

3. PATTERN ARCHITECTURE

Figure 4 shows an abstract architecture that capitalizes on the ability to create security enforcement patterns that separate local constraints from universal enforcement patterns. The architecture also enables automated policy translation and increased human understanding.

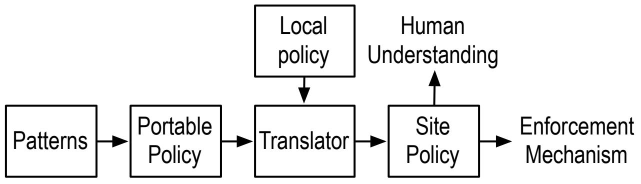


Figure 4. A system for applying patterns.

The patterns in the architecture represent the security engineering patterns that span organizations. Researchers have started applying pattern engineering to security [4][10]. Much of the research focuses on applied security software patterns, but many high level abstract patterns exist such as private data, or shared data. Engineering patterns arise from real world examples. However, the simplicity of the implementation space for enforcement policy allows for general descriptions of the pattern space.

Consider a model where every object has an owner. The system can share the object data in several ways. Figure 5 organizes these sharing patterns into an object-oriented hierarchy on a continuum of control.

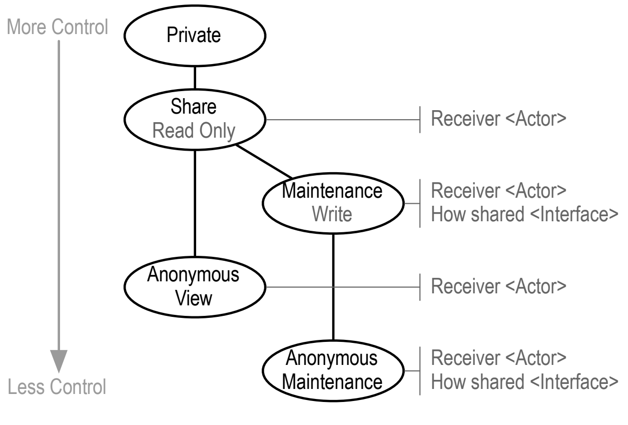
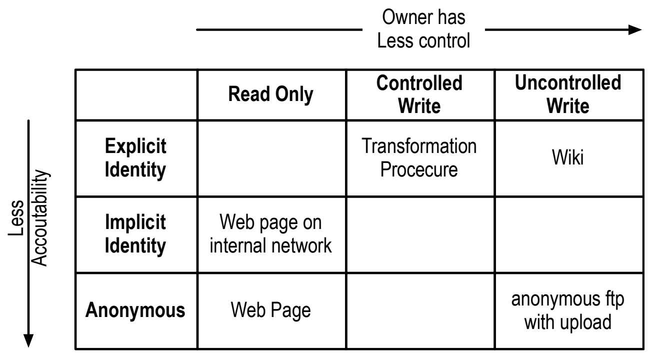


Figure 5. A simple object-oriented view of pattern space.

How strongly the policy designer can define the actor becomes an important factor in determining the control the owner or system retains over the data. Strong authentication yields the most control, anonymous the least control. However, a variety of interesting policies arise in the middle of the continuum. Consider a web server on an internal network. The web server allows anyone who has access to the network segment to read its web pages. The web server does not explicitly identify the users, but implicitly identifies them as all users that can access the network segment. Other examples of sets of implicit actors include all users who can access a host, or all users in an enterprise. Often these implicit actor sets open the door for human error by obscuring the impact of policy changes. For example, suppose only employees should access the internal web server. Suppose an outside auditor needs access to some internal resource. Without a clear understanding of all the implicit actor sets once the auditor gets the needed access to the network segment, they could erroneously receive access to the web server.

Similarly, experience has shown a second continuum for data sharing starting with: no sharing, read only, controlled write, and uncontrolled write. A controlled write represents a Clark-Wilson triple that controls "how" the actor modifies the object.

Figure 6 sketches a two dimensional view of the pattern space on the dimensions of control and accountability. Entries in the table show popular examples of the pattern in that portion.


Owner has Less control
Less Accoutability

	Read Only	Controlled Write	Uncontrolled Write
Explicit Identity		Transformation Procecure	Wiki
Implicit Identity	Web page on internal network		
Anonymous	Web Page		anonymous ftp with upload

Figure 6. Sketch of enforcement pattern space.

One could imagine creating additional continuums to make an n-dimensional space. Many parts of this pattern space represent undesirable security policies, but recognizing undesirable patterns represents the first step towards fixing them.

Patterns occur at different levels in an enforcement policy;

- Object class definitions – patterns of resource definitions.
- Actors – explicit, implicit as well as degree of trust organize actors into patterns
- Resources interfaces – patterns of how to access objects
- Constraints – a subset of resource interfaces that control who and when an actor can access the interface
- Trusted Procedures – patterns that control how the data can change.

Policy designers can combine these patterns into larger patterns that span all these elements. For example, an auditor role pattern implies a strong identified actor, read only access to the data, and append access to an audit log.

The architecture in Figure 4 incorporates these high-level pattern descriptions in the first stage. Application developers and vendors would then implement those patterns to create the specific policy specification for the target application. The vendor must list each of the resources the application consumes, and define them in a site independent manner to create the portable policy.

When the local administrators install the application they provide the necessary details for the local site. Using the portable policy and these local details the translator can produce the local site policy. Since the portable policy embeds the policy intent, a policy tool can show the local administrator the impact of proposed changes. Since the supported access control models are isomorphic, the translator can also produce specific enforcement mechanism policy for all the applications in the enterprise.

A prototype exemplar implementation using a similar architecture translated policy patterns written in java into a variety of enforcement mechanisms for the enterprise. Choosing java allowed policy developers to use a wide variety of existing object-oriented design tools. While many enforcement mechanisms have attractive graphical user interfaces, the exemplar showed the benefit of a language approach for sharing policy as with open source software. Given the assertion that creating enforcement policy exemplifies a complex design problem, approach that have worked for complex designs in the past should apply to security enforcement policies.

The exemplar showed that good patterns save engineering effort by clarifying thinking and provide easier mapping to requirements. This savings in effort and the resulting structure opens the door to do research into adding new capabilities to the enforcement policy to increase human understanding of the impact of changes. In particular, tools that show the impact of implicitly defined actors sets could clearly show the impact a malicious insider could have on an organization.

4. SUMMARY

Economics represents the biggest hurdle to developing high assurance systems. Technology that reduces the cost of developing secure systems will greatly improve the likelihood of getting solid security systems deployed. Every security system has an enforcement policy that developers must engineer. This paper provides several practical points that can reduce the cost for creating enforcement policies.

- Creating security enforcement policy has many of the complexities of software engineering and can benefit from many of same cost saving techniques, such as patterns and open source development.
- Effective security requires clarity and understanding.
- The expressiveness of different security models and translating between them requires further study to allow policy designers to pick the proper tool for the job and integrate policies from different models together.
- Creating enforcement policy must become part of secure system engineering. Evolutionary approaches to writing enforcement security result in higher cost and poorer polices.
- The intent of a security enforcement policy requires further research to ensure continued human understanding of the policy over its life cycle. It should include a model of what the system attempts to protect against.
- An open source enforcement policy approach can reduce the security costs for individual enterprises and better ensure the enforcement policy meets its intended purpose.

Overall, people need to recognize that creating enforcement policy represents a complex engineering discipline that needs further refinement. That shift in mindset can have a great impact on the quality and effectiveness of secure systems.

5. ACKNOWLEDGMENTS

My thanks to the Sandia National Lab cyber forum for creating a venue for discussion of critical computer security topics like policy management.

6. REFERENCES

[1] Boebert, W.E. and Kain, R.Y. 1985. A practical alternative to hierarchical integrity policies. *Proc. 8th National Computer Security Conference* (1985).

[2] Clark, D.D. and Wilson, D.R. 1987. A Comparison of Commercial and Military Computer Security Policies. *Proc. of the 1987 IEEE Symposium on Research in Security and Privacy*. (Mar. 1987), 184–194.

[3] Epstein, P. and Sandhu, R.S. 2001. Engineering of role/permission assignments. *Proc. 17th Annual Computer Security Applications Conference*. (2001), 127–136.

[4] Fernandez, E. et al. 2008. Patterns and Pattern Diagrams for Access Control. *Proc. of the 5th international conference on Trust, Privacy and Security in Digital Business*. (2008), 38–47.

[5] Gamma, E. et al. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Reading, MA.

[6] Lampson, B.W. 1973. A note on the confinement problem. *Commun. ACM*. 16, 10 (1973), 613-615.

[7] Neumann, G. and Strembeck, M. 2002. A scenario-driven role engineering process for functional RBAC roles. *Proc. 7th ACM symposium on Access control models and technologies* (2002), 33–42.

[8] Payne, C. et al. 1999. Napoleon: a recipe for workflow. *Proc. of the 15th Annual Computer Security Applications Conference* (1999), 134–142.

[9] Sandhu, R.S. et al. 2000. The NIST model for role-based access control: towards a unified standard. *Proc. 5th ACM Workshop on Role-based Access Control*. (Jan. 2000).

[10] Schumacher, M. 2006. *Security Patterns Integrating Security & Systems Engineering*. Wiley-India.

[11] Thomas, R.K. 1997. Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments. *Proc. 2nd ACM workshop on Role-based Access Control* (1997), 13–19.

[12] Thomsen, D.J. 2007. Patterns in Security Enforcement Policy Development. *International Conference on Database and Expert Systems Applications (DEXA)*. (2007), 744–748.

[13] Thomsen, D.J. and Haigh, T. 1990. A comparison of type enforcement and Unix setuid implementation of well-formed transactions. *Proc. 6th Annual Computer Security Applications Conference*. (Jan. 1990).

[14] Thomsen, D.J. et al. 1998. Role based access control framework for network enterprises. *Proc. 14th Annual Computer Security Applications Conference*. (1998), 50–58.

[15] Thomsen, D.J. et al. 1999. Napoleon: network application policy environment. *Proc. 4th ACM workshop on Role-based access control*. (1999), 145–152.