A User-Centered, Modular Authorization Service Built on an RBAC Foundation

Mary Ellen Zurko  
Iris Associates  
mzurko@iris.com

Rich Simon  
EMC Corporation  
simon_rich@emc.com

Tom Sanfilippo  
Groove Networks  
sanfilip@groove.net

Abstract

Psychological acceptability has been mentioned as a requirement for secure systems for as long as least privilege and fail safe defaults, but until now has been all but ignored in the actual design of secure systems. We place this principle at the center of our design for Adage, an authorization service for distributed applications. We employ usability design techniques to specify and test the features of our authorization language and the corresponding administrative GUI. Our testing results reinforce our initial design center and suggest directions for deployment of our authorization services. A modular architecture allows us to experiment with our design during short term integration, and evolve it for longer term exploration. An RBAC foundation enables coherent design of flexible authorization constraints and queries. We discuss lessons learned from the implementation of this service through a planned deployment in a context that must balance new research in risk management with dependencies on legacy services.

1. Introduction

The Authorization toolkit for Distributed Applications and Groups (Adage) project at The Open Group Research Institute explored authorization support in distributed environments. The primary goals were to make the expression and enforcement of computer-based authorization policy easy for system administrators and application writers, to support a rich variety of mechanisms built on a role-based access control (RBAC) foundation, and to use a modular architecture that made experimentation, integration, and deployment easier. To achieve these goals, Adage was based on several overarching design principles: user-centered design, policy-neutral design, modular architecture, and roles-based design.

From the beginning, the user-centered design of Adage focused on its potential users, the creators of authorization policies for distributed enterprises [49]. This is reflected in the design of Adage’s three interfaces:

- Special emphasis is placed on the design and usability of a graphical user interface (GUI) for policy definition and management. This GUI is constructed around basic concepts that are common to all environments requiring a complex yet consistent authorization policy, such as principals, objects, groups, and rules.
- Adage also provides an alternative to the GUI for power users, in the form of a textual Authorization Language (AL) for expressing policies. The GUI and AL are equivalent in power, but the AL provides a clearer path for customization and extensions.
- Adage provides a simple application programming interface (API) for developers to query the Adage authorization server.

The approach taken to designing mechanisms was pragmatic and policy-neutral, involving the review of many security models, policies, and principles [13, 5, 33, 20]. We distilled the material into a set of mechanisms that could support each of the policies and principles, as well as ad hoc policies that we observed in use. Chief among the principles supported were various forms of separation of duty [44, 24].

Authorization services are only part of a security infrastructure. We chose a modular approach to our architecture, that allows our authorization service to work with current and future authentication and attribute services, secure communication infrastructures, administrator languages, and database backends. Modularity within our service enables future enhancements to take advantage of new research directions and user requirements. This allows the integrator and deployer to make use of the best aspects of Adage while encouraging further research when new requirements are identified.

Throughout the design of Adage, role-based access control (RBAC) was an organizing and guiding principle. The enforcement engine is based on a combination of roles and rules for specifying and interpreting policy [44]. We were able to take advantage of existing work on RBAC models, extending them to cover the policies we studied. In addition, RBAC provides an excellent basis for answering the wide range of queries needed for both trouble shooting and auditing of site security. We did not explicitly include the notion of roles in our user interface, in order to keep the range of concepts to a minimum.

The remainder of this paper is structured as follows. We review related work in RBAC, user-centered design, and authorization. Next we give a detailed overview of the Adage architecture and authorization engine, emphasizing our modular architecture, RBAC foundations, and lessons learned from the implementation phase. We then discuss the user interface and usability testing of Adage, with an emphasis on the GUI. Results from the testing confirm many of our initial design choices and indicate directions for further work. We summarize our early experience deploying Adage and integrating it with other services and applications. We conclude by suggesting areas for future work and summarizing our experience.
2. Related work

Adage’s innovative authorization technology touches on related work along a variety of fronts. The four major themes in the design of Adage are support for a wide range of role-based access control models, a first-class treatment of usability issues, maintaining architectural flexibility, and providing a rich authorization modeling environment. Below we examine related work in each of these areas.

2.1. Separation of duty/RBAC

Much of the prior work in Separation of Duty (SoD) and Roles-Based Access Control (RBAC) was surveyed as part of the initial research for Adage. [44] contains a summary of the SoD and RBAC work that influenced the Adage architecture, as well as a presentation of a new SoD variant called History-Based Separation of Duty. The paper [44] also clarifies several of the different conceptions of SoD that exist, as well as various conceptions of roles and RBAC.

Adage implements a representation of roles as defined in the RBAC model of [42]. This work outlines a family of RBAC reference models, each of which provide different features and levels of expressiveness. Adage implements the most expressive of these reference models, termed RBAC₃, by including both role hierarchies and constraints. [42] also discusses the possibility of history-based constraints on RBAC systems that are similar to Adage’s history-based SoD variant, but the authors of [42] consider history-based constraints to be an advanced feature that lies outside their taxonomy of RBAC systems, due to the complexity of this feature. We found that history-based constraints can provide the fine granularity of control occasionally needed in models such as the Chinese Wall model. Also presented within [42] is an administration model for maintaining RBAC systems that parallels their taxonomy for general RBAC systems. The authors recommended separating administrative permissions from other application permissions, in order to model common separation of duty divisions between users and security administrators. This is also the approach taken by Adage, as is discussed below in the section “DARPA Deployment.”

Recently [24] presented a formal characterization of the SoD policies covered in [44], using a formal RBAC model. This allowed the exploration of issues encountered when composing SoD policies and the identification of classes of polices that are not composable. The formal model also supports the exploration of alternative implementations for SoD policies by providing a formal specification for such implementations.

[25] develops a new paradigm for reasoning about distributed access control. This paradigm, called DCM (Distributed Compartment Model), includes a role-based access control method that assumes users in a distributed system will need RBAC to distributed resources that span administrative boundaries. Adage also provides methods for sharing and reconciling role-based authorization information across administrative domains, but not at the level of granularity supported by DCM. The approach taken in Adage is to support the import and export of objects from administrative domains using an attribute authority facility, and to reconcile role information via a distributed trust model. Adage attribute authorities are described in the architecture section below, and the Adage trust model is described in the “Future Work” section. In contrast, DCM defines a detailed policy framework for managing dynamic changes in object ownership and administrative authority across administrative domains.

The RBAC model proposed in [23] attempts to bridge the gap between real-world roles in organizations and tradition RBAC support in database systems, especially as typified in [3]. Capturing real-world roles and policies is also a major goal of Adage. The approach taken in [23] is to provide tools for capturing and factoring user roles while engaged in the development of conceptual and logical database schema. This approach can be complimentary to the user-centered design and testing methods discussed below.

2.2. User-centered security

In the sections on “Usability in Secure Software” and “Directions in User-Centered Security” in [49], we discussed the history and progress of usability as a fundamental principle in secure systems. We also discussed security and privacy as issues in Computer Human Interface (CHI) work. There were notable early efforts [40] and several more recently [31, 21] that included usability in the stated goals of the secure system. Saltzer and Schroeder saw usability as a fundamental security principle, on a par with principles such as least privilege and fail-safe defaults. While these latter principles have become standards in the security literature, the user-centered principle has been honored more in the breach. Our work is the first that we know of that explicitly places security and usability as peer goals of an effort, and employs specific techniques to assure the achievement of both goals [7, 8, 9, 50, 51, 52]. The community’s emphasis on rigor has produced some excellent examples of authorization models with important and useful formal properties (as we discuss below). We believe that these approaches need to be married with the pragmatic user-centered methods we have documented elsewhere, in order to achieve systems that secure our information in both theory and practice.

2.3. Architectural approaches to authorization

Early implementations of authorization in operating systems and in shared and distributed applications built the authorization support directly into the system or application. That approach has the notable drawback that every system and application must re-implement an authorization methodology (thereby incuring a significant reliability burden), and each separately designed and implemented system worked differently. Later versions of
the Distributed Computing Environment (DCE) [39] improved on that basic approach by providing standard access control list (ACL) code that could be easily integrated into any DCE application. SnareWorks [14] moved ACL and other DCE services out of the application to a centralized server (accessed via a trap in a customized TCP driver that ensured the services would be called for all recognized TCP applications). CORBA [16] centralized the authorization decisions at the ORB, both for the ORB (solving the problem of how to be sure that authorization services are always invoked) and for security aware applications. Both the DCE and CORBA approaches require a particular infrastructure for interacting with other security functions such as authentication and auditing. HP Praesidium/Authorization Server [37] provides an authorization server that can be used by Web-based applications or applications that explicitly call it. It supports a wide variety of authentication mechanisms and can be extended to support new ones. It requires either DCE or HP/Praesidium/ Security Service for its infrastructure. The most recent research in authorization approaches, exemplified by SDSI, SPKI, and PolicyMaker [12, 38, 18], place authorization information such as identity, user groupings, and even policy rules into signed credentials that can be distributed at any time with any request. They are evaluated locally by an ACL-like mechanism or evaluation engine, along with whatever local policy information is available.

Our approach is most similar to Praesidium. We provide an authorization server that can be extended to use a variety of authentication formats and other distributed information. Our secure communications infrastructure can also be altered. The radically distributed approach of systems like PolicyMaker do not currently meet our user-centered security goals, as they do not enforce enough structure for the specialized views and interactions most suitable for ease of use [35]. In addition, users cannot query systems like that for full information about the accesses currently allowed in their system.

2.4. Authorization languages

Some of the earliest work on representation of authorization data is from [30], which lays out the access matrix model for protecting resources. The notion of ACLs developed from the sparse representation of this matrix as a list of tuples. The Adage language and engine retain the ability to express these simple tuple-based authorization relationships between subjects, actions, and objects. Most of the advanced Adage features are modifications and enhancements to this basic model. Statements in the Adage language are interpreted as definitions and constraints, and are evaluated algorithmically. In contrast, [4] represents authorization information as sentences in a first-order language. This logical representation of authorization data allows a model theoretic examination of policies and policy changes, as well as evaluation of access control decisions. In [27] a logical language is defined for explicitly modeling SoD policies and RBAC concepts. The formal nature of the language allows policy consistency and completeness questions to be investigated. Specifically, it allows policies to be examined for correctness. As noted in [42], logical languages are ideal for expressing the wide range of constraints that may be needed to implement complex SoD and RBAC policies. In contrast, the Adage system currently requires constraints to follow a set number of fixed schemas, though within a given schema, the administrator can combine a large and varying number of expressions and primitives. However, not every possible form of constraint has been included in Adage. For instance, temporal constraints are currently not represented. An interesting extension to the Adage AL would be to allow direct encoding of constraints using logical sentences referring to Adage objects. This would allow even more flexibility in Adage policy definition, while potentially simplifying the encoding of policies that are not easily captured by the existing constraint schemas. Modal operators could also be defined to allow the expression of temporal constraints as well.

3. Architectural overview

The Adage architecture is shown in Figure 1. Adage consists of a policy definition client and a policy decision server. The client contains the GUI (called Visual Policy Builder or VPB) and Authorization Language (AL) interpreter. The client communicates with the Authorization Decision Server (ADS) through the Administration API. In practice, the VPB and AL supplied by the Adage developers could be replaced with other clients. Authorization clients (those applications wishing an authorization decision) access the ADS through the Authorization API. These two APIs are currently abstractions implemented using CORBA IDL stubs.

The ADS stores policy information from policy definition clients in a database called the User Authorization Database (UAD). This database is in a form that permits easy display and manipulation of policy information by users. The ADS contains a translator that transforms the information in the UAD into a form more suitable for making rapid authorization decisions. This database is called the Engine Authorization Database (EAD). The authorization engine is the other major piece of the ADS. The engine uses the EAD to find rules that apply to a given decision.

The Adage architecture does not enforce any specific policy, but rather provides support for implementing a range of policies, including many common ones. The policy definition client may be used to define any kind of policy the user desires, and this policy is transformed into rules that the engine uses to enforce it.
In the design of the Adage architecture considerable effort was devoted to making the system modular and flexible, allowing it to be integrated with diverse application environments. Some of these environments will obviously depend on infrastructure technology different from that chosen for the initial implementation of Adage, and in some cases there will be outright conflicts. The Adage developers chose a client-server approach for dealing with this issue. Most of the Adage functionality resides in the ADS and is accessed remotely using communication mechanisms supported by Adage. Clients of Adage only need to access a thin client library that encapsulates communication with the ADS. Thus Adage management clients and applications using Adage services can be physically distributed relative to the ADS.

Internally, the ADS itself is highly modular. The RBAC engine that carries out rule evaluation is isolated from the AL interpreter by a translation layer. This division between the AL and the engine allows the support of alternative authorization languages, and also isolates most language changes to a layer well above the core algorithms of the system. This simplifies the design of the engine and better lends itself to formal verification [9]. Furthermore, the engine itself may be extended by adding more kinds of rules without requiring changes to other parts of the ADS. The split also aids the user interface design by allowing us to consider user interface choices separately from our underlying support. This helps us maintain our emphasis on usability instead of being driven by choices made because of particular back end features.

The modularity of the Adage architecture has already proven beneficial. The Adage client interfaces have been integrated with the Apache web server, the Tcl shell scripting environment, CORBA and COM applications, and third-party administration clients written in Java. All this has been done without modification to the ADS or communications interfaces. The ADS has used on commercial and non-commercial database technologies (including SQL Anywhere). Likewise, the Adage communications infrastructure has been adapted to run on CORBA ORBs, including omniORB. All these deployment options are supported without recompilation of the Adage sources, enabling Adage to be adapted to a wide variety of application environments with relative ease.

The remainder of this section discusses the client side, communications, and server side of our architecture in more detail.

### 3.1. Clients

Clients in the Adage architecture are categorized by the interface(s) they access. They can be Adage management clients or Adage applications, although any given client can be both.

For management clients we support two kinds of users: casual users and power users. The DoD is responding to resource pressures by targeting a decrease in the number of people administering computing infrastructure. This suggests that casual users who interact with Adage intermittently will be a significant portion of the user community. A GUI with on-line help provides the
casual user with constant reminders of context, state, and command. While the majority of users are likely to be casual, we also want to support power users who will learn the Adage features in detail and use them frequently. We expect power users to write extensions to our authorization functionality in the form of scripts that make common site-specific procedures quick and easy. These scripts could also allow sharing of customizations or policy information within an organization [32]. In the tradition of fourth generation languages [29] and domain-specific languages [17] our Authorization Language has constructs particularly suited to manipulating authorization constructs. Full specification and examples of the Adage GUI and Authorization Language can be found in [1].

The primary Adage management clients are the VPB, the AL Console, and the RALP (Remote AL Processing) interpreter. The VPB supports the definition, maintenance, and query of Adage policies from a GUI. Sophisticated clients like the VPB translate operations at the GUI into AL requests to the ADS, then interpret the results to update the user interface. The AL Console is an interactive window for entering AL commands and displaying their results. The RALP interpreter is a command-line executable for submitting batch scripts of AL commands to the ADS. It also supports local interpretation of Tcl commands in the batch input. These embedded Tcl commands are not transmitted to the ADS, however. The AL interpreter is designed to filter out general Tcl commands that might disrupt the operation of the server.

Adage application clients are applications that contact the ADS to request runtime authorization decisions. Decision requests may change the active state for users of the system. The active state of the Adage system is the collection of volatile runtime information that may affect the outcome of authorization decisions; the rules and role definitions are non-volatile in that they are only changed by management clients and never by access decisions themselves. An example of the active state information is the number of principals active in a role at the current time (the cardinality of the role).

The Adage AL is currently built upon the Tcl language environment [36]. We chose Tcl because it is an existing language that is widely deployed and easily accessible to administrators and researchers on many platforms. AL statements and commands are implemented as Tcl commands, and AL borrows some of its current syntax from Tcl syntax. However, the semantics of AL is independent of Tcl language semantics. AL is a declarative language for defining and performing operations on Adage objects. As such, AL does not make extensive demands on its host language environment. AL can easily be hosted by other language environments such as perl or Java, though interactively interpreted languages will usually provide some advantages for Adage administrators. Because AL is primarily a declarative language for representing at a high level the authorization policies enforced by the engine, it does not provide a mechanism for representing arbitrary authorization algorithms. In fact, AL is not a Turing-complete language [28]. AL does not currently have general computational constructs, or an extension mechanism for calling out to user-defined code. While this limits the power of AL, it also makes the system easier to analyze and verify.

3.1.1. User interface design and features

The features and functions presented by the graphical and textual user interfaces were designed using scenario-based techniques [15]. They support a number of documented authorization policies, such as Bell and LaPadula, Biba, Chinese Wall, a variety of role-based policies emphasizing separation of duty [5, 11, 13, 20, 24], and authorization policies based on ad hoc practices in industry and in open research environments. We present this information using a small number of unifying and simplifying constructs.

We chose groups as a simple and easy to understand relationship among the three basic entities: actors (principals or subjects), application actions (like read and write), and targets (accessible objects like files). Groups can be homogeneous or heterogeneous, and are represented differently in the GUI according to the type of entities in them (i.e., all groups of actors are displayed together). Groups can be explicitly defined (by placing each of the members of a group into that group) or implicitly defined via templates. Templates are specified using attributes (see below), and refer to all entities with the attribute values specified in the template, such as all targets with a label NOFORN. Any basic entity may be outside any group, in which case it is called a singleton.

Actors contain the set of principals (authenticated entities) that represent an individual. Providing a way to reference all the on-line incarnations of an individual encourages the formation of a consistent policy for all of the identities used by a particular person. Separation of duty policies require such a notion.

Attributes modify singletons or groups. We implemented secrecy and integrity levels (like those used for policies like Bell and LaPadula [5]), string-based labels (and sets of labels), and constraints that define an authorization requirement that must be satisfied. For instance, a constraint may preclude any overlap in membership between the current group and some other group (static separation of duty). A constraint may also specify the circumstances under which an action can be taken (such as user-secrecy-level >= target-secrecy-level).

Actions and targets are grouped separately from the actors who may be authorized to use them. The user interface contains no explicit notion of roles because we already had a large number of defined terms for the user to understand. However, rules associate a group of actors with groups of actions and targets and with a constraint or set of constraints that must be satisfied for access to be allowed. For instance, targets can be classified into several different (named) groups, and each group of targets can have different rules for access. This separation of access
permissions into a unit distinct from the users of those permissions is the hallmark of a role-based approach.

An Adage policy is a group of rules. We designed the Adage user interface to allow the administrator to try various policy alternatives in a “debugging” mode before actually deploying a policy or policy change. Adage supports active and inactive rules to allow the administrator to make changes and determine the effect of changes without affecting the security of the running system. Policies allow administrators to define potential authorization states (such as “active” and “testing only” or “war” and “peace”). Multiple policies can be defined and maintained independently. One of these policies is the active policy, and the rest are latent. Changes to any of these policies are immediately reflected in the underlying databases (see Section 3.3.1). Security administrators can build and test latent policies off-line, and then activate them when appropriate. A query capability allows authorization rules to be tested against both types of policies. Hypothetical tests are kept distinct from actual authorization decisions, even for the active policy. This is because real authorization decisions affect the active state of the system, which in turn affects future authorization decisions. A deep search facility allows administrators to examine relationships that may be hidden at a syntactic level of presentation. This helps in debugging, testing, and manually back-tracing authorization decisions.

3.2. Communications

The client-server architecture of Adage assumes that clients contact the Adage servers responsible for an Adage cell using any protocol supported by the server. An Adage cell is an administrative domain in which all clients share a common policy, and there is one logical ADS. Implementations may need to replicate ADS instances for efficiency. Within an Adage cell, a logical ADS may be in communication with several authentication and attribute authorities for the exchange of authorization information with other environments. Examples of authentication and attribute authorities include Single Sign-On systems, Key Distribution Centers, directory servers, PKI registration authorities, and web server access control databases. Interfaces with these authorities may involve periodic batch-oriented exchanges of AL statements, or real-time on-line communication. Authorities may reside in different Adage cells or in the same Adage cell as the ADS they are communicating with. When located in remote Adage cells, the information provided by the authority may be less trusted than locally available information.

The above communications architecture is the logical architecture of the Adage system. Currently Adage implements only a subset of these features. For instance, ADS instances are not replicated. Also, there have been no attribute authority interfaces implemented to date, though a PKI registration authority has been designed. The ADS currently supports only the CORBA IIOP protocol, or IIOP over SSL when used with an ORB toolkit that supports SSL. Future versions of the ADS could easily be modified to support multiple protocols, since the ADS is a multi-threaded server. At the client side, alternative client libraries have already been constructed for interfacing COM applications to the Adage client libraries.

3.2.1. Management and decision interfaces

The ADS provides two interfaces; a management interface and a decision interface. A major design choice was the representation for AL commands and results as they were being transmitted over the management communications interface. It was decided to represent the transmitted AL as a Tcl command stream, and all results as Tcl command output. Additionally, when possible, the output of an AL command was also a valid AL command, in order to facilitate re-use of the output.

An alternative to using Tcl command streams was to devise a representation for all the AL objects, commands, and results using CORBA IDL datatypes or some other network data representation. However, past experience has shown us that such systems can be large and difficult to maintain. Using discrete CORBA datatypes would have resulted in a very large CORBA-specific code base for the communications implementation, while providing little benefit beyond strong type checking. By using Tcl pipes for the management interface, Adage clients and servers can more easily support alternative communications toolkits and protocols in the future.

The client libraries and corresponding CORBA IDL interfaces export a small number of methods, so integrators typically only need to make one or two function calls to request an Adage decision or management operation. Because the client libraries do not implement any of the Adage system semantics, sophisticated integrators can make direct calls to the Adage CORBA IDL interfaces. The flexibility and simplicity of the client interfaces allows applications to be integrated quickly. Most of the integration burden is in mapping the application’s objects into Adage objects, making sure the Adage client libraries are compatible with the application runtime environment, and setting up a secure authenticated communications channel with the ADS.

The Adage decision interface is accessed via another CORBA IDL. The caller provides the principle, application action, and target for the authorization request, and the ADS returns a boolean result. We provided a simple runtime interface to ease integration and deployment. More complex decision interfaces will be needed in the future to support task based models, to allow applications to mark the beginning and ending of tasks [47].

Further information on the Adage management and decision interfaces, including programming API documentation, is available in the Adage Design Specification [2].

3.3. Server

The primary components of the ADS are the AL interpreter, the translator, the RBAC engine, the UAD and
EAD database interfaces, and the server side of the communications interfaces.

3.3.1. Databases and their translator

Adage uses two logically distinct databases to maintain persistent information for Adage policies. The main benefits of this separation are the isolation of the policy enforcement mechanisms from the policy definition mechanisms, improved engine performance, and simplified engine design. Segregation of the UAD and EAD makes the system more modular and flexible, as well as more efficient. The UAD is used to store Adage policy objects defined through the AL interpreter, while the EAD stores a compiled form of the AL definitions that is optimized for evaluation by the EAD engine. Since the UAD is designed primarily to support the AL object model, new administrative features, such as improved search facilities, can be added with relative ease and without adding additional database overhead to the engine, jeopardizing its efficiency. The EAD in turn supports a compact, portable representation for evaluation of Adage authorization rules, that does not lend itself to user-level query. Engine design is simplified because the engine need not support the user level niceties that are required for a good interface (like hierarchical groups). This form is optimized for searching for rules that pertain to a given decision. It also reduces the abstract, hierarchical representation in the UAD to concrete, flat representation. By maintaining a compact, portable representation, the engine and EAD can potentially support alternative deployment models, such as replicating the EAD or portions of the EAD to different machines. This deployment model for Adage is currently being explored by members of the DARPA IA program discussed in the section “DARPA Deployment” below.

Both the UAD and EAD are isolated from their physical storage facilities by a database independent API. In the case of the UAD this API is a Tcl library for accessing objects in the data model supporting the AL. Within the library, commands for retrieving and manipulating AL objects are translated to SQL queries and passed to an ODBC database independent interface. This allows the UAD to be hosted on a number of different databases, both commercial and non-commercial. Early versions of Adage were tested with the freely available MSQL database. The latest version supports Sybase SQL Anywhere, Microsoft SQL Server, and Personal Oracle Lite. Other ODBC compliant databases can also be supported. The use of standard database APIs also allows the storage for the UAD to be physically separated from the ADS. While this may be a drawback from an assurance perspective, it would allow a version of Adage supporting multiple instances of the ADS to be more easily constructed, since third-party database replication technology could be brought to bear. Also, by using a SQL database for the UAD, the addition of new objects to the AL data model is greatly simplified. The use of a SQL database also allows for the possibility of exploiting other database features such as transactions, referential integrity, and on-line backup, in support of the UAD. We experimented with a direct translation of the AL object model into a logical database schema. This feature may have been valuable in allowing users of Adage to query the SQL-based schema directly. However, the inefficiencies introduced by the direct translation required that the Adage entities be consolidated. Note that this is a common problem with many SQL-based systems.

The EAD is also accessed and maintained through an abstraction layer. When the ADS is running, the entire EAD is loaded in to memory for efficiency, and any changes are propagated to a persistent representation incrementally (this design may produce scalability problems later that will have to be addressed). Changes to the EAD, which propagate through the AL interpreter and translator, are logged in real-time to the EAD persistent store. This store is organized as a transaction log, which records state changes to the in-memory copy. By logging a copy of the EAD state changes, the ADS can start up more quickly and recover more quickly from failures. Without the EAD persistent store, the in-memory copy would need to be constructed by re-translating all the AL definitions stored in the UAD. The current implementation of Adage uses a text file for the EAD persistent store. An EAD persistent store based on the ObjectStore PSE OODB has also been designed.

The translator supported incremental changes, so that a change to a policy in the UAD made using the VPB could be reflected immediately in the EAD; it was not necessary to translate the entire UAD again. This made interactive policy definition simple.

3.3.2. Adage authorization engine

The user-visible policy building mechanisms in the VPB and AL are implemented using platform-independent architectural mechanisms. These architectural mechanisms are embodied in the Adage authorization engine. The engine is the portion of the ADS that actually computes access. Each access decision involves a principal (authenticated entity), an application action, and a target. The engine receives information about the principal, action, and target in the form of labels that contain the names of roles pertinent to the decision. The engine selects the rules relevant to the decision by examining the labels, and then applies each rule. If any rule fails or if no rules are found that pertain to the decision, access is denied. Role-based access is enforced by labeling users, targets, actions, and rules with the names of roles with which they are associated. This is done by the translator. The names of roles are derived from the names of the groups used in a rule definition; see “User Interface Design and Features” above. When an access decision is required, the engine is given the labels of the entities involved and uses them to determine what rules to apply.

Adage as a whole implements a very general representation of roles, corresponding to the RBAC model in [42] The Adage policy definition interface allows both
role hierarchies (through the generic hierarchical grouping mechanism) and constraints. The roles are used to label rules that apply to access decisions involving the roles, but the relationship of one role to another is not conserved at the enforcement level. Another discussion of the Adage Authorization Engine can be found in [44].

3.3.2.1. Rules

A role at the engine level is a triplet of entities: a set of principals (a team), a set of actions (an action set), and a set of targets (a collection). Each role has a corresponding set of zero or more rules that determine what the role is allowed to do. In order to have any access at all, a principal must be a member of some team; a principal that is not a member of any team may not act in any role and so has no authorized access to any action or target (fail-safe default). If a principal is in a team, then the set of rules corresponding to the role determines what the principal may do when acting in that role. In the simplest case where there are no explicit rules corresponding to the role, the principals in a role’s team are allowed to perform any of the actions in the action set to any of the targets in the collection; the implicit rule in this case is simply that a principal must be a member of a role’s team to act in the role. The more interesting cases arise when constraints involving the labeling of targets and principals, or the relationship of roles to one another, or of the principal with respect to an object, may modify what the principal may do in the role. In these cases, labels are used to select the set of rules that apply to a given decision.

3.3.2.2. Labels

Labels are used to mark principals, targets, and actions with the roles that are relevant to access decisions. In most cases, the roles in the label are just those to which the entity belongs. The roles form a non-hierarchical set of names. The engine label may also contain hierarchical confidentiality and integrity levels.

Target and action labels are straightforward. Both contain the names of the roles in which the target or action is used. Principals have two kinds of labels: a label closure and a current label. The label closure is exactly like the target and action labels; it contains all the roles to which the principal belongs. The current label contains the roles that the user has active; this is a subset of the roles in the label closure. Allowing the principal’s current label to contain more than a single role is an important usability feature since it mirrors how people use roles in actual job situations [33, 41].

The Adage label design is similar to that suggested in [41] which points out the importance of keeping distinct the notions of user, principal, and process (called a subject in the cited paper). The problem is avoiding unnecessary constraints on how people use their roles while still preventing insecure information flow. In the cited paper, user’s labels are allowed to float “up” the label hierarchy, gathering greater privilege, while principal and process labels remain fixed.

At the VPB level, actors are the surrogates of users and each user has one or more principals, each principal being associated with one user. An administrator defines the set of labels over which a user may range by assigning the user’s actor or individual principals to a role (as represented by team and its association with a set of rules). While the actor may range over all permissible values (since it represents all the user’s principals), principals may be more constrained by assigning them to only a few or one role. Assigning a principal to one and only one role would be equivalent to a policy in which principals have fixed labels. In such a policy, a user would have to authenticate as a different principal for each role. The ability to allow principals to have more than one permissible label opens up more flexible and easier to use policies. At the architectural level, a principal’s label floats up as more roles are activated, within the permissible range defined by the policy for each principal.

One problem with allowing principal labels to float is that targets created by the principals would have labels that contain too many role names if they inherited their labels from the creating principal. Adage applies the label of the action that creates the target to the new target. This means that any role having access to that action has (potentially) access to the new target, and this set of roles will be some subset of the roles in the principal’s current label.

3.3.2.3. Rules

The constraints defined at the user interface level are transformed into rules for use by the engine. Engine rules contain the expression of policy at the architectural level. An engine rule contains the constraint that must be enforced, and labeling information that determines when the rule should be applied. The labeling information for a rule is called the rule’s scope.

There are two kinds of rules: activation rules used when a user is entering a role, and non-activation rules that control the use of the role once it has been activated.

In the simplest case, a user activates a role by performing an action in the role’s action set. In more complex cases, the activation may require other constraints to be satisfied. These constraints would be expressed in rules about the mutual exclusion or cardinality of the roles.

While the user interface enforces static cardinality, the engine can enforce a dynamic cardinality constraint. A dynamic cardinality constraint specifies the number of users that may be active in a role at one time. The engine maintains a counter for each role that tells the number of users active in the role. If a user tries to activate a role having a dynamic cardinality constraint, the engine checks if the role currently has the maximum number of users.

The engine maintains a set of information called the active state that contains the volatile information required to make an access decision. The active state contains the current cardinality of each role and the current labels of each principal. This information is currently protected by a single-threading lock to avoid concurrency issues.
Adage supports two simple forms of separation of duty. One of these, static separation of duty, is enforced by the administrative commands and not the engine. Static separation of duty merely requires that the roles share no members. This can be determined at policy definition time and requires no runtime support.

Simple dynamic separation of duty requires runtime support. In this form of separation of duty, two roles may share members but no user may have both roles active at once. This kind of constraint is expressed by using an activation rule to enforce the mutual exclusion of the roles.

Adage supports a variety of constraints that compare the hierarchical and non-hierarchical portions of principal and target labels. They may be used to express policies like Bell and LaPadula or Biba.

3.3.2.4. Computing access

The engine selects rules based on the roles involved in the access decision and the scopes of the rules. The roles involved in the access decision are computed from the principal, target, and action labels that are input to the engine. This set of roles is compared to the scope of each rule to determine if the rule is relevant to the decision being made. If no rules are found that pertain to a decision, access is denied. During the rule selection, the engine determines if granting access will require some roles to be activated. The activation rules for those roles are selected and tested. If the rules pass, then the roles may be activated, and the remaining rules are checked. If all of these rules pass, then the access is granted and the principal’s current label is augmented by any roles that had to be activated.

4. Usability testing

Our emphasis on the usability and flexibility of Adage drove both the architecture and the features presented by the user interface. We applied user-centered security [49] techniques at the design phase, then tested the early results with two forms of usability testing. While we considered both security administrators and application writers to be users, we refer to the security administrator interface in this section.

4.1. Usability testing of the authorization GUI

Simply stating that a user interface is “user friendly” does not supply any proof that the statement is true. One way to ensure the friendliness of a user interface is to perform usability testing [34]. We ran two types of usability tests on Adage: one to provide long term information on potential directions in security management tools, and one to provide short term feedback on our GUI and the view it offers of authorization information. The specific techniques we implemented are low-cost usability techniques, involving a high payback with limited investment of time and money. Full details of our testing and results are available [50, 51, 52].

For the broader usability testing we chose a form of contextual usability testing [10]. Contextual techniques involve visiting users in the context in which they do their work. They aim to capture a rich view of the problem, and are good for discovering aspects of the problem previously unknown to the testers. They do not produce detailed measurements. We used a form of contextual interview. The structure of the interview is not rigid; the interviewer is encouraged to follow-up any direction that looks profitable. Our interview questionnaire included asking about the user’s background, job responsibilities, organizational culture, work habits and processes, security policies, and explicit scenarios [50].

To balance the free form contextual interviews, we did formal usability testing that allows us to get detailed feedback on the particular model and user interface we were implementing. In a formal test, users interact with an early prototype of our GUI and try to perform tasks assigned by the tester, for example adding a new user with appropriate controls. We record their reactions and feedback while trying to accomplish those tasks with a think-aloud protocol. We also measure their overall experience with an exit questionnaire. The goals of this formal testing are to verify that simple tasks can be easily executed, gauge the satisfaction of novice users with the initial implementation of our GUI, document novice user errors and confusion, and analyze the results and recommend changes. The four tasks we asked the user to accomplish in a one hour time slot were removing a user from the system, adding a new user with appropriate controls, servicing a complaint about lack of access to resources, and designing an implementation of a policy to control the release of information.

4.2. Usability testing results

We ran the tests with five subjects with experience managing security for distributed applications. They were split between a research organization and a government laboratory.

4.2.1. Contextual interview results

We were able to get a good picture of the nature of a security administrator’s job [51]. This helps us verify our current design and suggests future directions for user-centered support.

- It is unobtrusive. As one subject said, “No one notices what you do, but you get your head cut off if there’s a problem.” Another stated that the nature of the job is such that if no one complains, you are doing your job.
- It is dynamic. Subjects mentioned quickly shifting responsibilities, fighting fires, and juggling queues of people and problems to be solved.
- It is cooperative. Several subjects discussed security problems they had worked on that involved multiple people with different areas of expertise, different responsibilities, and sometimes different locations.
• It is learning-oriented. Several subjects indicated they enjoyed their job because it gives them a chance to try out new technology. Another subject indicated that he liked reading “Practical Unix and Internet Security” [22] because it had “lots of cool low level stuff.” “Reading that book gets you paranoid.” While this aspect indicates the potential for a well-informed user base, their ability and willingness to read should not be used to preclude core usability work.

Adage incorporates several features to support the dynamic and cooperative aspects of system administration. Features that support inactive subjects and policies allow tasks that are partly done to be put on hold without losing state. Our named grouping and labeling support the ability to mark any group of singletons or groups with a free form textual name. Our subjects found this to be so helpful for communicating intentions and history during the formal testing that one of them suggested an additional free text annotation field be available on all entities. This would allow administrators to leave each other electronic notes in the same manner as the self-adhesive, removable paper notes. It’s tempting to consider more heavyweight solutions such as integrating existing groupware products or protocols, especially given the administrators’ interest in learning new technologies. This would need to be balanced against their need for rapid turnaround and control. One subject described himself as a “flat files” kind of person, preferring the efficiency of vi or emacs over other more structured management tools.

Security experts often voice concern about the lack of knowledge administrators have about security issues. Our study shows that administrators are actively interested in learning about technical issues like security, both through hands on local experimentation and through reading. Security products continue to overlook these avenues. One subject discussed his experience with logging tools. He knew they sent alerts to root, and he hoped that a security violation was an alert, but he really wasn’t sure. Security products for intrusion detection don’t give the security administrator much of a chance to learn through examples, unless they’re run in conjunction with tools that attempt well known attacks (a risk not all administrators are willing to take with a running system). Adage can facilitate that learning about authorization policy by shipping example authorization rules and databases that can be inspected and queried while not applying the policy to the current state of the system.

One subject’s discussion of how he set the authorization policy for his DCE and Web service indicated the difficulty in managing security and remaining unobtrusive. There was no pre-written policy (the norm for all the subjects we interviewed), so he knew he would have to define it as he went. He started with a simple policy that he expected to be too restrictive (internal users had full access, external users had none). He tried to set a policy that was fairly tight so that it could be loosened over time, as he knew a loose policy would not get tightened up. Research in access control that can learn from accesses, point out patterns, and suggest adaptation might allow administrators to maintain control over time while minimizing the number of complaints.

During contextual interviews, we asked our subjects about the sorts of tools and interfaces they find useful. The subjects all looked favorably on GUI tools. However, several subjects related instances when GUI tools were inadequate. For instance, Subject 1 could not rely on a GUI ACL editor tool to change access control information on a firewall machine. Getting familiar with the command line tool took some time, and he would have preferred to use the GUI. Subject 4 uses the DCE command line tools for everything because the GUI tools they were using were not available on all platforms. The GUI tool also could not be tailored to manipulate information they had added, and it lagged behind the command line tool in supporting product features. While advances in secure cross platform GUI support such as Java can overcome some of these problems, solutions for others rely on further research on GUI user programming.

4.2.2. Formal usability testing results

The two outputs from the formal usability testing were notes on the subjects’ actions and statements, and the numeric results of the exit questionnaire. The notes were interpreted and categorized using affinity mapping techniques [10]. Full details of these results are in [52].

In usability testing of this sort, no documentation or on-line help is provided. We are testing the utility and understandability of the system itself. Previous experience with testing generic computer users (non-administrators) indicated that people were comfortable with this approach and that it was the way they would naturally approach the application. However, our administrators were different. In the contextual interviews most of our subjects explicitly missed having either a paper manual or on-line help.

An authorization interface’s first job is to make the policy implemented through it clear. Three of the assigned tasks asked the administrator to make changes to an existing organization’s policy with which he was unfamiliar. Each of the subjects mentioned the need for some way for them to become acquainted with the existing policy information more rapidly than the ability to browse through items by type, expand the groupings, and view the details on any selected items. Users needed an overview of all the connections between the groupings as well as views of the entities filtered on particular attribute values. Subjects found it difficult to reason about groups within groups. As a result, we discarded the notion of using virtual groups with more complex relationships (“This group contains all the members of group A who are not also members of group B”). Several subjects also indicated the need for a feature that would compare two entities (like rules or actors) and show the differences. For example, one subject wanted to do a full comparison between the accesses allowed to a user that has access to a particular target and a user that had complained that she did not have access to that target.
Most subjects wondered how rules would combine. We explained that if any rule precluded access, it would be denied (a logical AND of the rules). The subjects found this to be straightforward and understandable. Some were still concerned that an administrator could make a rule that would override an existing rule without knowing it, such as denying access to a particular person when that access is explicitly granted to one or more groups that that person belongs to. This problem has been researched in the authorization community [6]. Our design center is that every active set of rules represents a single, consistent authorization policy. We initially designed our system to notify the user of such conflicts and suggest rules that clearly stated the resulting policy (“Everyone in group G except for Mary can read target T”). Such a feature would require application-specific information about which actions conflict with each other. (Fortunately, we did not have time to experiment with this feature.)

One of the thrusts of our research has been to try to find ways of allowing administrators to specify authorization policy about targets without being tied to their namespace, but still allowing them to take advantage of the natural groupings in their namespace, by using features such as wildcarding. We attempted to emphasize the use of homogeneous groups of targets called collections, which could include specific targets, other collections, and target templates (groups of targets specified by attributes or with wildcarding in the names). Three of our subjects specifically, spontaneously and enthusiastically stated that they liked collections. However, it remained difficult for them to consider forming policies about groups of targets without relying on names. One assigned task was specifically designed to get subjects thinking in terms of labels and collections. Some subjects considered a labels-based approach, but pointed out that accurately labeling all the targets would be difficult, time-consuming and error-prone. One subject suggested that he could use a tool that could determine if a target was mislabeled based on semantic matching. Other subjects considered a more traditional namespace approach; marking one or more directories “external” and moving all external targets to those directories. Collections can be used to do this without rearranging the namespace, yet our users did not suggest this approach. The namespace based approach to access control is in line with the current scenarios outlined by several subjects during contextual interviews. Another subject considered the hybrid approach of labeling by directories. We continue to believe that collections can facilitate desired policies more easily, while recognizing that we have not managed to communicate that vision with our current user interface.

One of our tasks was structured so that the most obvious response would run afoul of a constraint on user groups that could not overlap (static separation of duty constraints on roles). All of our test users were unfamiliar with the concept, but some understood it immediately from the error message that the user was already a member of a conflicting group. All of the users expressed confusion about the best way to proceed within an unknown organization context. Two of the subjects considered alternatives that circumvented the constraint, and would have given the user in question the requested accesses associated with one group while being a member of the conflicting group. Sandhu [42] discusses some of the issues around deciding at what granularity to enforce conflicting roles. This experience indicates that systems should allow administrators to explicitly indicate the level of enforcement that should be maintained across changes over time.

The subjects were asked to evaluate the GUI representation of a variety of concepts on a scale from obvious (1) to confusing (6). The results, arranged from most to least obvious, were:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Average Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu Commands</td>
<td>2</td>
</tr>
<tr>
<td>Targets</td>
<td>2</td>
</tr>
<tr>
<td>Users</td>
<td>2.2</td>
</tr>
<tr>
<td>Groups</td>
<td>2.4</td>
</tr>
<tr>
<td>Policies</td>
<td>2.6</td>
</tr>
<tr>
<td>Actions</td>
<td>3</td>
</tr>
<tr>
<td>Constraints</td>
<td>3.25</td>
</tr>
<tr>
<td>Roles</td>
<td>3.4</td>
</tr>
<tr>
<td>Labels</td>
<td>3.6</td>
</tr>
</tbody>
</table>

These numbers indicate that while all of these concepts were communicated with some clarity to our users, actions, constraints, roles and labels were challenging enough to warrant further study.

Subjects were asked to rate two statements from strongly agree (1) to strongly disagree (6):

- “I was able to easily discover what items (users, groups, rules, etc.) were in the database” rated 1.4.
- “I was able to easily find items I was looking for” rated 2.6.

This indicated that browsing was fairly intuitive, but that searching was less so. This prototype of our GUI did not have the search command implemented; it was emulated by the tester when the subjects asked for it.

Finally, subjects were asked to rate their interaction in general on a scale from very satisfying (1) to very frustrating (6). The average of this rating was 2.8, a modestly positive initial experience. One subject echoed our hopes and plans for the prototype that was tested, “This is not yet a finished, polished product. I’m sure that your user tests will help—it’s something missing from many products!”

5. DARPA deployment

As part of the DARPA Information Assurance Program [26], Adage has been enhanced for deployment within the government’s Next Generation Information Infrastructure (NGII). This work was started under the Pledge project at The Open Group Research Institute [43]. The Pledge project provided an opportunity to explore both the strengths and shortcomings of the current Adage
An administrative facility was designed for segregating the development and maintenance of policies from the activation and deactivation policies. This facility enforces access control on the modification of AL definitions stored within Adage. Under the access control model, certain Adage teams have administrative authority over projects. Project administrators can build new rules for their projects and for objects within their projects. Another class of users, designated policy administrators, can define new projects, assign project administrators to projects, and choose policies to activate or inactivate. The policy administrator can look over all the projects that have contributed rules to a policy before deciding to activate the policy or not.

In general Adage did not focus on the high assurance aspects of designing and implementing a distributed authorization system, other than the attention give to the usability features and the careful design of the engine, as noted above. In contrast the Pledge project began the process of hardening the actual implementation by providing for communication of Adage over an SSL transport and supporting certificate-based authentication of users and servers. However, more work is needed in this area, such as securing the connections to the UAD and EAD physical storage. In terms of on-disk privacy, the use of commercial SQL technology such as Sybase SQL Anywhere allows this data to be maintained in an encrypted form. A similar facility is still needed for the EAD physical store, however.

[42] presents several models for administration of RBAC definitions, and Adage in fact implements some of the recommendations for segregation of permissions outlined there. However, we have yet to classify the Adage access control mechanisms according to the specific models of RBAC therein. Such an analysis would be an excellent task for a future project.

In the course of their work for the IA program, Secure Computing Corporation (SCC) implemented transparent authorization for a CORBA application using the ADS interfaces, and a set of programming tools they developed in Java. SCC’s NAPOLEON [48] tool allows application developers to read in CORBA IDL files and identify CORBA objects and methods that will later be referenced in authorization rules. It then directly exports these objects and methods as targets and action definitions for inclusion in the Adage UAD. Adage administrators then create rules for these objects. At run-time, SCC installs a custom interceptor in the target application that makes calls to the Adage runtime decision interface, providing the names of the objects and methods that are being invoked by the application. Adage in turn provides decisions on the actions requested by the application. SCC’s interceptor framework thus allowed Adage to be integrated with COTS CORBA applications, which is a major focus for the IA program. Additionally the ability to integrate with COTS COM and web server applications is also presently available. The integration of Adage with SCC’s Napoleon tool was valuable because it validated both the flexibility of the Adage architecture, and the flexibility of the Adage rule system, since the integration was affected with no modifications to the Adage system, and only minimal modifications to the Napoleon tools.

In May 1998, Adage was selected to be the standard policy language and decision engine for the entire IA program. Though Adage was designed for distributed authorization management, it was envisioned that it could be extended to encompass other types of policies relevant to the program, such as intrusion detection and authentication polices. Several proposed enhancements were identified as part of the integration with SCC, such as the ability to group actions using an object container model. Currently the SCC model for using Adage AL maps CORBA methods and classes to targets and collections. The only action defined is “invoke.” This mapping was required because Adage does not currently support groups of actions. This is because in the development of Adage the was a strong emphasis on only including features that were necessary to model the real-world policies we encountered during scenario-based design. Our scenarios did not capture the need for a hierarchical distribution of actions. Had action groups been implemented, a more natural mapping for CORBA authorization would have been to create action groups for classes and actions for individual methods. This would leave the target namespace available to identify individual object instances.

6. Future work

We have pursued a variety of new ideas that expanded on our initial architectural work in modularity and integration and on our user-centered authorization approach. We outline this future research agenda here.

Our work defined the most general version of separation of duty, history-based separation of duty. In history-based separation of duty, the specific history of a user with the target is used to determine how a user may act in a role. This form of separation of duty does not prevent a user from activating a role, so it is not expressed with an activation rule. Instead, rules are used that apply constraints on which actions in a role may be performed on a target based on the specific history of the user with that target. For a simple example, imagine two roles, Order Creator for creating purchase orders, and Order Approver for approving purchase orders. Roles of this kind are usually constrained by some form of separation of duty. A static constraint would simply prevent anyone from being in both roles. This is often an impractical or overly rigid solution. A simple dynamic constraint would allow someone to act in both roles, but not at the same time. This is better, but still not as flexible as some real world policies. In a history-based constraint, a person
could act in both roles simultaneously, but could never
Approve an order that they had previously Created. This
allows greater flexibility and is more consistent with how
such policies are treated in businesses. [44] discusses
history-based separation of duty in more detail, including
elements of history-based constraints.

Version 1.0 of Adage, completed in June of 1998, did
not implement the history-based constraints. Our plan was
to focus on more fundamental mechanisms in v. 1.0 and
implement history-based constraints as part of a following-
on project. We note here that implementing history-based
constraints must address a number of complex design
issues, among them:

- Representation of history. History is conceptually a
  three-dimensional bit matrix, having axes representing
  principals, actions, and targets. A bit in a given
  position indicates whether or not that principal has
  performed that action on that target. This conceptual
  model is unacceptable as an implementation model;
  the amount of information required is too vast. A
  workable history design requires some efficient repre-
  sentation of this sparse matrix, as well as some way to
  represent pending access (see next bullet).

- Accuracy of history. The fact that the Adage
  enforcement engine grants access to a target does not
  mean that that access actually occurs; it may fail to
  happen for reasons other than security (resource
  limits, programming errors, etc.) At what point does
  the history get updated to reflect that the access has
  happened? It seems that some kind of two-phase
  commit procedure is required; the history is first
  updated to represent a pending access, and revised
  later when the access has actually happened (or has
  failed to happen, in which case the history is cleared).

- Longevity of history. How long does history remain
  valid and how much history is kept? If a principal,
  action, or target is deleted, does the corresponding
  history get deleted?

The conflicting desires for use of mobile code and
safety on the Internet led the Adage project to investigate
authorization policy as a way to control how systems and
networks are used, and to investigate ways to detect and
respond to intrusions automatically. A follow-on project
called POEM (Policy Enforcement for Mobile Code)
sought to enhance and apply the Adage group authoriza-
tion server technology to the challenging problem of
malicious or faulty mobile code attacking or damaging a
host system. To make mobile code safe while retaining its
potential usefulness, mobile code must be securely
confined within flexible walls that allow enough freedom
to do legitimate tasks while limiting malicious or faulty
activity. However, any policy permissive enough to allow
useful work will contain some potential for abuse.
Therefore, a second line of defense is needed and is
provided by an intrusion detection and response system.

Unfortunately, intrusion detection technology is still
very young and current detectors yield many false posi-
tives, making any automated response to the alleged abuse
problematic [45, 46]. Even following a true intrusion,
automated response can easily lead to premature or
excessive denial of access to resources, effectively causing
the detection system to contribute to a denial of service
attack. The central goal of POEM was to provide
measured response to intrusions. A set of intrusion metrics
would characterize an intrusion with respect to kind,
severity, and certainty. These intrusion metrics could be
used to define contingent policies tailored to a particular
context. For example, the kind and severity may be used to
define specific responses to specific attacks (e.g., “Server
A is experiencing a port scanning attack.”).

Finding ways to reference a large number of diverse
targets remains an issue. There are techniques for dealing
with large namespaces that can be used with the current
Adage implementation, such as using a logical target name
for a whole class of targets recognized by an application,
but this obviously makes the individual targets invisible to
the system, and thus limits the policies that can be
expressed. A general approach to defining an arbitrary
collection of targets, such as a regular expression syntax
for matching target names, could be pursued.

However, a better approach would be to allow expres-
sion of authorization policies using the conventions of the
native namespace supported by the actual targets of the
application. For instance, policies referring to objects in an
object-oriented target namespace, could use mechanisms
like inheritance to group objects. These concepts could
also be applied to managing actions and groups of actions.
Similarly, policies on a relational data store would refer
objects native to the relational database model, such as
tables, views, and statement permissions. Much work has
been done on defining federated authorization models for
federated namespaces and federated databases [39, 19].
Most of these models attempt to develop a common syntax
for representing names across varying kinds of
namespaces. In contrast, the approach suggested for Adage
is to support the installation of custom namespace modules
that would act as extensions to the AL to allow expression
of Adage policies using the namespace syntax and group
reference mechanisms of the native application. The AL
interpreter, and possibly the engine, would make call-outs
to the namespace modules to compute static constraints and
 closures.

Since our authorization server is using external informa-
tion as part of its decisions, and is designed to function
in a highly distributed world where there are many
information sources and scaling is important, it needs a
trust model that specifies how trustworthy an information
source is, what information that source is allowed to
provide, and which authorization decisions that source is
allowed to influence. Referrals are necessary for the server
to make effective use of new sources and to scale well.
This work is more complex than existing work in authenti-
cation trust models, as their output is generally a simple
indication of whether or not some piece of authentication
information should be trusted. We developed such a trust
model [1], but did not implement it.
Translating Adage user level constructs to the access control mechanisms used by applications such as file systems and firewalls would allow an administrator to maintain a single, consistent view of the authorization policy of all her applications in an easy to use manner. There are technical issues of how to translate to different lower level mechanisms and propagate changes to them. In addition, an infrastructure is needed to specify the limitations of what kind of access control can be specified for each of these backends, and new user level constructs are needed to indicate the inevitable variability in back end specifics.

Our design center and usability test subjects are enterprise security administrators. Designing authorization support for a wider range of end users is a more challenging goal. The advent of personal Web servers and mobile Java code may mean that many more users will need to specify and manage their security policy. Research is needed on radical alternatives to enabling users to secure their resources according to personal policy.

7. Conclusions

Our initial goals were to prototype an authorization service for use with distributed applications whose primary emphasis was on the usability of its administrative interface. The service was designed with a modular internal architecture that allowed it to plug in to a number of distributed infrastructures. We used an RBAC foundation to provide policy-neutral authorization support and user-centered features such as rich queries and separation of duty. Our implementation experience shows that separating user interface and runtime authorization concerns works to the benefit of both, enabling new user features that do not slow down or jeopardize efficient application authorization decisions. This same separation allows exploration of alternative front ends (as with SCC’s NAPOLEON) and alternative backends (output to different databases or legacy applications).

A small number of user interface concepts can be powerfully and flexibly combined to support a wide range of real world policies. Usability testing confirmed that our use of groups, attributes, constraints and rules could be meaningfully manipulated by a novice user in under one hour, with no documentation. Our simple initial model of security administration as unobtrusive, dynamic, cooperative, and learning-oriented will continue to drive new features and directions in supporting tools. A variety of future directions are presented, most importantly integration with mobile code and intrusion detection, and referencing large numbers of targets that are themselves referenced by applications using heterogeneous namespaces.

We were unable to continue and extend our research at The Open Group Research Institute because its board voted to close down a number of internal organizations, including the Research Institute. However, the DARPA deployment program accepted BBN’s proposal to extend Adage’s Authorization Language to specify policies for security services beyond access control.

Acknowledgements

This work would not exist without the efforts of the rest of the team: Marty Hurley, Greg Carpenter, Rick Cormier, Ed Childs, Liza Martin, Shui Fun Poon, Travis Broughton, Rich Ford, Nimisha Mehta, and Alim Needham. We also thank Teresa Lunt, Gary Koob, and Sami Saydjari for sharing our vision and funding Adage and Pledge and Mary Denz for overseeing those contracts. Thanks also to our anonymous usability testing subjects for giving their time and talent, to our anonymous reviewers for their informative and careful feedback, and to Denise Simon for her editorial help.

Bibliography


