On the Unification of Access Control and Data Services

David Ferraiolo[1], Serban Gavrila[1], Wayne Jansen[2]

[1]National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{dferraiolo, gavrila}@nist.gov

[2]Bayview Behavioral Consulting
Point Roberts, WA 98281, USA
jansen@computer.org

Abstract

A primary objective of enterprise computing (via a data center, cloud, etc.) is the controlled delivery of data services (DS). Typical DSs include applications such as email, workflow, and records management, as well as system level features, such as file and access control management. Although access control (AC) currently plays an important role in imposing control over the execution of DS capabilities, AC can be more fundamental to computing than one might expect. That is, if properly designed, a single AC mechanism can simultaneously implement, control, and deliver capabilities of multiple DSs. The Policy Machine (PM) is an AC framework that has been designed with this objective in mind. This paper describes the PM features that provide a generic AC mechanism to implement DS capabilities, and comprehensively enforces mission tailored access control policies across DSs.

Keywords: Access Control; Data Services, Access Control Policy, Policy Machine, Operating Environment

1. Introduction

Controlled delivery of data services (DS) is a primary objective of enterprise computing. In addition to the ubiquitous electronic mail and file management, DSs commonly deployed in an enterprise include services for time and attendance reporting, payroll processing, health benefits management, and workflow management. All DSs include computational capabilities that consist of operation/object pairs, which allow the reading, writing, and management of data, and distribution of access rights. Control over the delivery of DSs is achieved through authentication and access control (AC) mechanisms, typically made available through an operating environment (OE).

While AC currently plays an important role in securing DSs, by building AC and DS from the same underlying elements, AC can serve an even more substantial role in computing. With this goal of unification in mind, the Policy Machine (PM)'s AC framework was designed to implement a security critical portion of the program logic of arbitrary DSs and to enforce arbitrary, mission-tailored AC policies over DSs, solely through the configuration of its AC data [8]. PM policies are attribute based and multiple, distinct classes of policy can be maintained under the framework. The type of an object is transparent to users—they can view and consume all data regardless of type, in a manner consistent with the defined policies, and under a single authenticated session.

To appreciate the PM's benefits in computing, it is important to recognize the methods in which DSs are delivered today. Each DS runs in an OE, which can be of many types (e.g., operating systems, database systems, and many applications), each implementing its own routines to enable the execution of DS-specific operations (e.g., read, send, approve, select) on their respective data types (e.g., files, messages, work items, records). To impose control, each OE typically implements its own method for identifying and authenticating its users. In addition to authentication, many OEs implement their own method of AC to selectively limit a user's ability to perform operations on its objects.

The heterogeneity among OEs introduces a number of administrative and policy enforcement challenges and user inconveniences. Administrators must contend with a multitude of security domains when managing access policy, each with a local scope of control (user, data), and ordinary users and administrators alike must authenticate to and establish sessions within different OEs in order to exercise legitimate DS capabilities. Even if properly coordinated across OEs, access control policies are not always globally enforced and leakage or other unwanted access can occur. An email application may, for example, distribute files to users regardless of an operating system's

protection settings on those files. Moreover, special types of controls that are required over sensitive data can be especially difficult to implement in a piecemeal fashion across different OEs.

To alleviate these security management, policy enforcement, and usability challenges, the PM offers a multi-user, enterprise-wide OE. The approach taken is to provide a generic AC mechanism that can implement computational capabilities of arbitrary DSs, and displaces fractional AC decision making and enforcement by different OEs, with a single administrative domain and scope of control, amounting to a general purpose OE.

The PM has evolved from a concept, to a formal specification [9], to a reference implementation and open source distribution, which have served as a basis for an ANSI/INCITS standardization effort under the title of "Next Generation Access Control" (NGAC) [1, 2]. Previous publications [6, 7] have described the PM's capabilities in expressing and enforcing a wide variety of AC policies. The focus of this paper is on the PM's unification of AC and DSs.

The remainder of the paper gives in increasing details to the PM's approach to achieving the unification.

2. Overview

At its highest level the PM is a logical "machine" comprising:

- Two types of objects: AC data consisting of data elements and relations used to express access control policies and DS capabilities, and DS data;
- A set of operations: read, write[1], for DS data, plus administrative operations for configuring the AC data; and
- A set of functions for trapping and enforcing policy on access requests, for computing access decisions to accommodate or reject those requests based on the current state of the AC data, and for automatically altering access state when specified access events occur.

The PM is based on the premise that AC and DS logic can be expressed in terms of the same elements. AC provides the underpinnings for unification. DS capabilities are delivered to users through AC requests and policy is enforced over those requests, but only with respect to the operation and object types of the OE in which the AC is implemented. The question is whether a single AC can be general enough to support the operation types and object types of arbitrary DSs? To accommodate arbitrary DS object types and operation types through a single AC mechanism, the PM takes a data-centric approach. That is,

the PM does not control access to DSs, but to DS data types (e.g., documents, messages), which are treated simply as objects that can be read and written[2]. DS operations (e.g., read, send, submit, approve, schedule) are treated as combinations of read/write operations on DS data and administrative operations on AC data that alter the access state. In addition, the PM organizes the AC data and DS data using three kinds of containers, which are instrumental in the distribution of capabilities to users and the expression and enforcement of policies.

Some aspects of DS functionality cannot be accommodated by the PM. For example, operations such as spell checking, font selection, and user presentation, pertain to specific methods for writing and reading and must be implemented in DS logic outside of the PM's purview.

The PM is a generic OE in the sense that through the same access request interface, set of operations, AC data elements and relations, and functional components, arbitrary DSs can be delivered to users. Arbitrary access control policies can be expressed and enforced over executions of DS capabilities. Because the PM's functional architecture is fixed, the PM's AC data elements and relations provide the basic ingredients for expressing policy and DS logic.

2.1. Elements and relations

The PM's AC data elements include users, processes, objects, and three kinds of containers: user and object attributes, and policy classes. These containers group and characterize their members to reflect vital traits relevant to policy and DSs. User containers can represent roles, such as doctor or bank teller; affiliations, such as divisions, communities-of-interest or teams; or other common characteristics pertinent to policy, such as security clearances. Processes, through which a user attempts access, take on the same attributes as the invoking user. Data object containers work similarly in characterizing data, by identifying collections of objects such as those associated with certain projects, applications, or security classifications. Object containers can also represent compound objects, such as folders, inboxes, table columns or rows, to satisfy the requirements of different DS's. Policy class containers are used to group and characterize collections of policy or DS at a broad level, with each container representing a distinct set of related configuration items.

An assignment relation provides the linkage between all configuration elements and the properties they represent. For example, the assignment of a user to a user

---

[1] Our unification approach is based on a slight specialization of NGAC, as we limit resource operations to read and write, and NGAC does not.

[2] Execute operation is not included because it is an indirect method of reading and writing data which can be controlled by the PM.

attribute denotes that the user obtains the properties held or represented by the attribute. The same convention applies to the assignment of objects to object attributes and the other pairwise assignments between configuration elements allowed by the relation. Through assignments to a designated policy class, attribute containers and the elements they contain are organized into distinct classes of policy or DS. Policy classes can be mutually exclusive or overlap to various degrees to meet a wide range of policy and DS requirements and styles of administration.

Two other key attribute-oriented relations help represent a PM policy and DSs: associations and prohibitions. Associations describe the access rights that a class of users holds over a class of data objects or AC data, which are used to authorize operations that can be performed. Prohibitions describe access rights that users or their processes cannot exercise over a class of data objects or AC data, which are used correspondingly to prevent operations from being performed. When deciding whether to grant or deny an access request, an authorization decision function evaluates privileges and restrictions that are derived respectively from all of the prevailing associations and prohibitions.

One final PM relation is obligations. Each obligation stipulates a sequence of actions that are to be taken when an associated, pre-designated, access event takes place. Obligations serve as a way of automating administrative actions and representing certain types of dynamic conditions, such as those involved in separation of duty or workflow policies.

2.2. Functional architecture

The PM functional architecture (vis. figure 1) involves several components that work together to bring about policy preserving access and DSs. Among these components is a Policy Enforcement Point (PEP) that traps DS access requests. An access request includes a process id, user id, operation, and object. The operation may be a read/write, or administrative and the object may be a DS data element or an AC data element or relation. Administrative operational routines are implemented in the Policy Administration Point (PAP) and read/write routines are implemented in Resource Servers (RS). To determine if a request should be granted or denied the PEP submits the request to a Policy Decision Point (PDP). The PDP computes a decision based on current configuration of data elements and relations stored in the Policy Information Point (PIP), via the PAP. The PDP returns a decision of grant or deny to the PEP. If access is granted, and the operation was read/write the PDP also returns the physical location where the object's content resides, and the PEP issues a command to the appropriate RS to execute the operation on the object content, and the RS returns the status. In the case of a read operation, the

RS also returns the data type of the object (e.g., .ppt.) and the PEP invokes the correct DS application for its consumption. If the request pertained to an administrative operation and the decision was grant, the PEP issues a command to the PAP for execution of the operation on the data element or relation stored in the PIP, and the PAP returns the status to the PEP. Users access data of varying data types, in a manner consistent with the defined policies, and under a single authenticated session. If the status of successful is returned by either the RS or PAP the PEP submits the context of the access to the Event Processing Point (EPP). If the context matches an event pattern of an obligation the EPP automatically executes the administrative operations of that obligation, potentially changing the access state.

Note that the AC is data type agnostic. It sees objects as just data or AC elements and relations, and it is not until after the access process is complete, that the actual type of object matters to the DS.
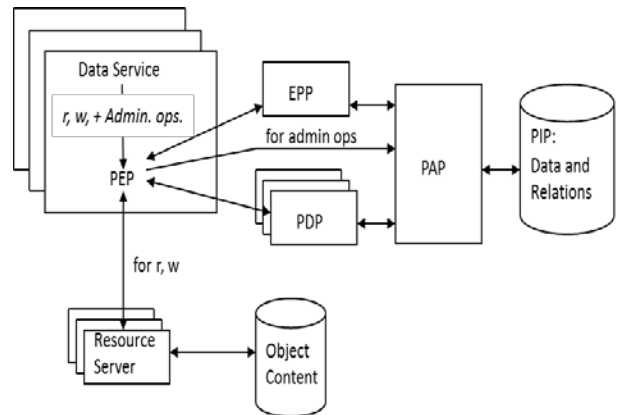

Figure 1. PM functional architecture

3. Specifics and examples

User attributes and object attributes generically characterize their members. An object is a name that indirectly references its content. As such, every object is an attribute of itself. Through assignments to a designated policy class, attribute containers and the elements they contain are organized into distinct classes of policy. Every user, user attribute, and object attribute must be contained in at least one policy class. Policy classes can be mutually exclusive or overlap to various degrees to meet a wide range of policy and DS requirements.

We use the symbol "→" to generically denote these assignment relations regardless of their type without danger of confusion, in that assignment always infers containment. We denote by "→$^+$" a chain of 1 or more assignment relations.

In addition to assignments, PM includes an association relation. An association is a triple, denoted by *ua---ops---ce*, where *ua* is a user attribute, *ops* is an operation set, and *ce* is a configuration element, comprising of either a user attribute, object attribute, operation set, or policy class. Its meaning is that the users contained in *ua* can perform the operations contained in *ops* on the objects contained in *ce*.

Figure 2 illustrates two example assignment and association relations depicted as graphs – one an AC policy configuration with policy class Project Access, vis. figure 2a, and the other vis. figure 2b, a DS configuration with File Management as its policy class. Users and user attributes are on the left side of the graphs and objects and object attributes are on the right. The arrows represent assignment relations and the dashed lines are associations.
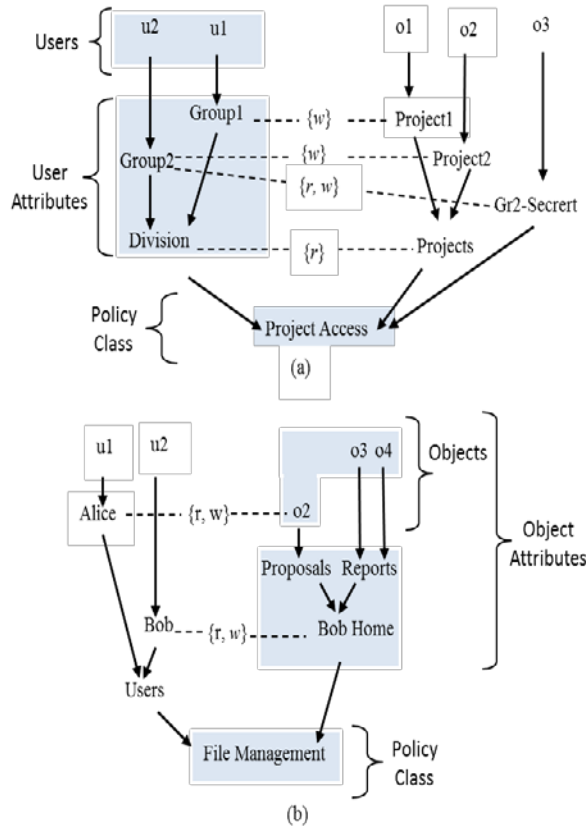

Figure 2. Two example assignment and association graphs

The AC of Figure 2a specifies that users assigned to either Group1 or Group2 can read data objects contained in Projects, but only Group1 users can write to Project1 data and only Group2 users can write to Project2 data. The Policy further specifies that only Group2 users can read/write data objects in Gr2-Secret. While figure 2a specifies policies for how its data can be read and written, the configuration is considered incomplete in that it does not specify how its elements and relations were created and can be managed. Figure 2b depicts a File Management DS. User u2 (Bob) has read/write access to objects assigned to folders (Proposals and Reports) that are assigned to his home directory (Bob Home). The configuration also shows user u1 (Alice) with read/write access to object o2. This configuration is also considered incomplete in that one would expect user capabilities to create and manage folders and to create and assign objects to their folders. Another feature common to File Management is the capability for a user to grant access to objects that are under his/her control to other users.

We specify these and other management capabilities for the Project Access policy and File Management DS, later in this section.

Collectively associations and assignments indirectly specify privileges of the form (*u*, *op*, *o*), with the meaning that user *u* is permitted (or has a capability) to perform operation *op* on object *o*.

PM includes a combining algorithm for defining privileges with respect to policy classes. Specifically, (*u*, *op*, *o*) is a privilege iff for each policy class, *pc* in which *o* is contained there exists an association *ua---ops---ce*, such that *u* is in *ua*, *op* is in *ops*, and *o* is in *ce* and *ce* is in *pc*. Note the algorithm also works when there is just one policy class.

The left and right columns of table 1 list the privileges for the graphs (a) and (b) of figure 2, when considered independent of one another. Table 2 lists the privileges for these graphs in combination. Note that (*u1 r, o1*) is a privilege in table 2 because o1 is only in policy class Project Access and there exist an association Division---{*r*}--- Projects, where u1 is in Division, r is in {r}, and o1 is in Projects. Note that (*u1, w, o2*) is not a privilege because o2 is in both Project Access and File Management, and although there exist an association Alice---{r, w}---o2, where u1 is in Alice, w is in {r, w}, and o2 is in o2 in File Management, no such association exists in Project Access. As we later demonstrate this combining algorithm provides the basis for comprehensive policy enforcement over DSs.

Table 1. List of derived privileges for the independent configurations of figures 2(a) and 2(b)

(*u1, r, o1*), (*u1, w, o1*), (*u1, r, o2*), (*u2, r, o1*), (*u2, r, o2*), (*u2, w, o2*), (*u2, r, o3*), (*u2, w, o3*)	(*u1, r, o2*), (*u1, w, o2*), (*u2, r, o2*), (*u2, w, o2*), (*u2, r, o3*), (*u2, w, o3*), (*u2, r, o4*), (*u2, w, o4*)

Table 2. List of derived privileges for the combined configurations of figures 2(a) and 2(b)

(*u1, r, o1*), (*u1, w, o1*), (*u1, r, o2*), (*u2, r, o1*), (*u2, r, o2*), (*u2, w, o2*), (*u2, r, o3*), (*u2, w, o3*)

In reality, users do not actually issue access requests; processes do, usually, but not always, on behalf of a user. In addition to users, PM includes processes among its basic elements. The PM treats users and processes as independent but related entities. A process is a system entity, with memory, and operates on behalf of a user. A user may be associated with one or more processes, while a process is always associated with just one user. The function *process user*(*p*) denotes the user associated with process *p*. We denote by <*op*, *o*>*p* a process access request, where *op* is an operation and *o* is an object.

PM includes two types of prohibitions: user-deny and process-deny. In general, deny relations specify privilege exceptions. We denote a user-based deny relation by *u_deny*(*u*/*ua*, *ops*, *os*), where *u*/*ua* is either a user *u*, or a user attribute *ua*, *ops* is an operation set, and *os* is an object set. Its meaning is that user *u* or any user assigned to ua cannot perform the operations in *ops* on the objects in *os*. User-deny relations can be created by an administrator or as a consequence of an obligation. For example, an administrator could impose a condition where no user is able to alter his/her own Tax Return, in spite of the fact that he/she is assigned to an IRS Auditor user attribute with capabilities to read/write all tax returns. When created through an obligation, user-deny relations can take on a number of dynamic policy conditions to include that of separation of duties (if a user executed capability x, that user would be immediately precluded from being able to perform capability y).

A process-deny relation is a triple of the form *p_deny*(*p*, *ops*, *os*), where *p* is a process, *ops* is an operation set, and *os* is an object set. Its meaning is that the process *p* cannot perform operations in *ops* on the objects in *os*. By specifying *os* as its complement, denoted by ¬, the meaning of *p_deny*(*p*, *ops*, ¬*os*) is that the process can not perform the operations in *ops* on objects not in *os*. Process-deny relations are exclusively created through obligations. Their primary use is in the enforcement of confinement conditions (if a process reads Top Secret data, preclude that process from writing to any data object container other than that of Top Secret).

With derived privileges and prohibitions in place, we are now able to describe the PM authorization decision function. The PM authorization decision function governs accesses in terms of user processes. When determining whether to grant or deny an access request, the authorization decision function takes into account all privileges and restrictions that apply to a user and its processes, which are derived from relevant associations and prohibitions, giving prohibitions precedence over privileges. That is, access requests to perform an operation on an object are issued only from processes acting on behalf of some user, and are granted authorization only if the processes' user holds appropriate privileges that allows the access and no restriction exists that would prevent the access. Otherwise, the access request is denied:

A process access request <*op*, *o*>*p* is granted iff there exists a privilege (*u*, *op*, *o*), where *u* = *process user*(*p*), and no restriction (*u*, *op*, *o*) or (*p*, *op*, *o*) exists.

Administrative operations are implemented through parameterized routines, prefixed by a condition, with a body that describes how a data set or relation (denoted by R) changes to R'. The condition tests the validity of the actual parameters. If the condition evaluates to false, then the routine fails:

```
Rtnname (x₁, x₂, …, xₖ) {
    if {conditions} then
        {
        R₁'= f₁ (R₁, x₁, x₂, …, xₖ)
        R₂'= f₂ (R₂, x₁, x₂, …, xₖ)
        …
        }}
```

The remainder of this paper uses administrative operations whose meaning should be obvious from their names. To execute an administrative operation the requesting user must possess the required capability. Just as capabilities to perform operations on data objects are defined in terms of associations, so too are capabilities to perform administrative operations on data elements and relations.

Recall that an association has the general format: *ua --- ops ---* ce, where *ua is* a user attribute, *ops* is a set of operations, and *ce* is an object attribute, policy class, user attribute, or operation set. The exact treatment/interpretation of *ce* depends on the operations included in *ops*, but in general the association specifies that users of *ua* are authorized to perform the operations in *ops* on elements in *ce*.

The following association enables administration of the assignments depicted in figure 2a, by users assigned to ProjectAccessAdmin (not shown):

ProjectAccessAdmin --- {*create-u*, *delete-u*, *create-ua*, *delete-ua*, *create-o*, *delete-o*, *create-oa*, *delete-oa*, *r*, *w*}--- Project Access

Its meaning is that all users in ProjectAccessAdmin can create and delete users, user attributes, objects and object attributes, as well as read and write objects, anywhere in the policy class Project Access.

The next two associations enable users in ProjectAccessAdmin to create and delete associations in the Project Access policy class:

ProjectAccessAdmin -- {*create-assc-from*, *delete-assc-from*} -- Division
ProjectAccessAdmin -- {*create-assc-to*, *delete-assc-to*}, Projects

To prevent a user assigned to ProjectAccessAdmin from giving him/herself access to resources controlled by

Project Access policy, a different administrator could create the following prohibition:

u-deny(ProjectAccessAdmin, {*r*, *w*}, Projects ∪ Gr2-Secret)

The question remains, how are administrative capabilities created? The answer begins with a *super user* with capabilities to perform all administrative operations on all data elements and relations. A super user can either directly create administrative capabilities or more practically can create administrators and delegate to them capabilities to create and delete administrative privileges. PM provides a single administrative domain and enables a systematic approach to the creation of administrative roles and delegation of administrative capabilities, beginning with a super user and ending with users with DS capabilities.

In addition to their individual executions, an authorized administrator can execute a collection of administrative operations through an administrative command. An *administrative command* is a parameterized sequence of administrative operations.

Consider the following administrative command within the context of figure 2b, where the user attribute Users assigned to the File Management policy class pre-exists:

```
create-file-mgmt-user(user-id, user-name,
user-home) {
  create-ua(user-name, Users);
  create-u(user-id, user-name);
  create-oa(user-home, File Management);
  create-assc(user-name, {r, w}, user-
    home);
  create-assc(user-name, {create-o,
    delete-o}, user-home);
  create-assc(user-name, {create-ooa,
    delete-ooa,create-oaoa, delete-oaoa},
    user-home);
  create-assc(user-name, {create-assc,
    delete-assc},{Users, {r, w}, user-
    home}); }
```

Through the execution of this command with parameters (*u*1, *Bob* and *Bob Home*), the user attribute "Bob" is created and assigned to "Users", and user *u*1 is created and assigned to "Bob". In addition, the object attribute "Bob Home" is created and assigned to policy class "File Management". Through the execution of this command user u1 is delegated capabilities to create, organize, and delete object attributes (DS folders) in Bob Home. In addition, *u*1 is provided with capabilities to create, read, write and delete objects that correspond to files and place those files into his folders. In addition, *u*1 is provided with capabilities to "grant" to users in the "Users" container, capabilities to perform read/write operations on individual files or to all files in folders in his Home. As indicated, by figure 2b User u2 (Bob) has granted user u1 (Alice) read/write access to object o2.

It is important to recognize that regardless of the method in which a DS distributes capabilities, the Project Access policy will always be comprehensively enforced, so long as no user of those DSs can delete o→⁺ Gr2-Secrets assignments for any object o. For instance, although the current configuration of the File Management DS gives user u2 (Bob) the capability to grant Alice read/write access to object o3, Alice would not be able to execute that capability (in accordance with the authorization decision function and in consideration of the Project Access policy). As another example, imagine an email DS that includes the association Charlie---{r}---CharlieInbox, where email users have the capabilities to assign objects to each other's Inboxes. As an email user, Bob could assign o3 to CharlieInbox, but unless the Project Access policy authorized Charlie to read o3, the assignment would have no effect.

Final examples pertains to obligations, which consist of a pair (*ep*, *r*) (usually denoted **when** *ep* **do** *r*), where *ep* is an *event pattern* and *r* is a sequence of administrative operations, called a *response*. The event pattern specifies conditions that if matched by the context surrounding a process' successful execution of an operation on an object (an event), the administrative operations of the associated response are immediately executed. The context may pertain to and the event pattern may specify parameters like the user of the process, the operation executed, and the attribute(s) in which the object is contained.

Obligations can specify operational conditions in support of history-based policies and DSs. Such conditions include Conflict-of-Interest (if a user reads information from a sensitive data set, that user is prohibited from reading data from a second data set), Work Flow (approving (writing to a field of) a work item, enables a second user to read and approve the work item). Also, included among history-based policies are those that prevent leakage of data to unauthorized principals.

Consider the cumulative configuration for the Project Access Policy and File Management DS once again. Although Bob cannot successfully provide Alice read access to object o3 through his grant capability, Bob could still provide Alice with the ability to read the content of o3. This could be achieved by Bob first reading the content of o3 and then writing that content to o2. Even if we were to trust Bob not to perform such actions, a malicious process acting on Bob's behalf could, without Bob's knowledge. To prevent this leakage we add the following obligation to our configuration:

When any process p performs (r, o) where o→⁺ Gr2-Secrets **do** create p-deny(p, {w}, ¬Gr2-Secrets)

This obligation will prevent a process (and its user) from reading the contents of any object in Gr2-Secrets and writing it to an object in a different container (outside of Gr2-Secrets).

A typical OE DS element is copy/paste. Also typical of copy/paste is the issue that two processes could cooperate in leaking data through the use of this feature. That is, one process could read data, e.g., a Gr2-Secrets file, followed by a copy/paste operation from the memory of the first process to the memory of the second process followed by the second process writing the data to another object, making the data available to users that are not authorized to read the data. We can accommodate copy/paste while addressing this concern. The copy/paste functionality can be implemented as DS specific logic (not provided by the PM) and two PM administrative commands; one for copy and the other for paste. The copy command is executed with the capabilities of the user running the first process. The DS logic reads highlighted text from the source object into a clipboard. The copy command creates an object that represents the clipboard. This event triggers an obligation response that assigns the clipboard object to all attributes of the source object. The paste command is executed with the capabilities of the same user running a second process. The paste command attempts to read the clipboard object. If the paste command is successful, the DS logic inserts the clipboard content into the second process' memory at a designated position.

The copy/paste functionality provides the expected inter-process communication, while adhering to policy. The PM enabled logic (copy and paste administrative commands) and the previous obligation, together prevent copying of an object in Gr2-Secrets and the subsequent pasting of its contents into an object that is not in Gr2-Secrets. That is, the copy operation would create and assign the clipboard object to Gr2-Secrets, and in accordance with the previous obligation, any subsequent process that reads from the clipboard (e.g., paste) would be prevented from writing to any object (e.g., o2) that is not in Gr2-Secrets.

4. Related work

Previous publications [6, 7] have demonstrated the PM's ability to separate policy from mechanism, by describing the PM's capability in expressing and enforcing combinations of well-documented policies, including role-based, discretionary, and mandatory access controls, as well as support for novel types of policies that have been conceived but never implemented due to the lack of a suitable enforcement mechanism. The approach applied the same functional architecture and AC data elements and relations defined in this paper, but did not specify an approach for delivering DS capabilities.

The notion of a multi-policy machine capable of combining policies was envisioned by Hosmer in the early 90's [10]. Inherent to our approach to comprehensive enforcement, is the ability to combine policies. However,

the two approaches are quite different. The multi-policy machine combines and resolves conflicts among multiple access control policies each implemented inside a different access control mechanism. The individual mechanisms that compute decisions and enforce policy are assumed to be already implemented and may pertain to a variety of policies. The ultimate decision to grant or deny a request in the multi-policy machine is based on metapolicies (order, priorities, etc.) and a voting schema.

Our approach is based on the PM that requires changes only in its configuration in the enforcement of arbitrary and organization specific attribute-based access control policies. Some of these policies happen to be composed of combinations of sub-policies. Besides being a single (enterprise-wide) mechanism, the PM does not resolve conflicts, and does not use metadata for combining policy elements.

The Extensible Access Control Markup Language (XACML) is an XML-based language standard designed to express security policies, as well as the access requests and responses needed for querying the policy system and reaching an authorization decision [11]. XACML is similar to the PM insofar as it provides a flexible, mechanism-independent representation of policy rules that may vary in granularity, and it employs attributes in computing decisions. Unlike the PM, XACML does not provide a single OE or offer a method for accommodating DS logic. An XACML deployment consists of multiple OEs that share a common policy decision function. Each of these OEs implements its own method of authentication, operational routines, and types of objects. Requests are issued from, and decisions are returned to, an OE specific PEP through a standardized request and response language. XACML and the PM also differ in their approach to creating and altering its AC data. As a prominent feature in its approach to unification of AC and DSs, the PM manages its AC data through a standard set of administrative operations, applying the same PEP interface and reference mediation function as it uses for accessing data resources. XACML does not recognize administrative operations, but instead manages policy content through its language, and offers no administrative method for the management of its attributes.

The concept of trust management as an overlay on existing OEs was introduced with the PolicyMaker system [3]. Trust management presents a comprehensive approach to specifying and interpreting security policies, credentials, and relationships to allow authorization decisions to be made about security-critical actions. To provide a coherent framework for expressing interrelationships between security policies, security credentials, and trust relationships, PolicyMaker and its successor KeyNote utilize public key infrastructure environments, certificate-based trust, and binding of cryptographic keys to actions [4, 5]. Unlike other access

control approaches focused mainly on OEs, trust management is particularly suited for situations where security policy is decentralized and distributed across a network, such as multi-system applications and DSs that cross departmental and organizational boundaries [5]. In this regard, PolicyMaker and KeyNote are similar to our use of the PM in providing a general purpose policy management and enforcement approach in support of DSs, but does not attempt to implement the capabilities of those DSs.

5. Conclusion

In this paper, we suggest that the next level of evolution for AC lies in a unification of AC and DSs. The PM was designed with this evolutionary goal in mind, amounting to a general purpose OE. The PM is a generic OE in the sense that through the same access request interface, set of operations, AC data elements and relations, and functional components, arbitrary DSs can be delivered to users, and arbitrary, mission-tailored access control policies can be expressed and enforced over executions of DS capabilities, solely through the configuration of its AC data. The practical benefits are many. Rather than a user having to authenticate to multiple OEs to exercise legitimate DS capabilities, a user can access all of his/her data, regardless of its type, in a manner consistent with policy, under a single authenticated session. This is because the PM is data type agnostic, and offers a single OE and scope of control. Policies are also globally enforced over DS, due to the PM's combining algorithm for deriving privileges and global treatment of prohibitions when computing authorization decisions. Rather than Administrators having to contend with a multitude of OE specific security domains when managing access policy, the PM provides a single administrative domain and enables a systematic approach to the creation of administrative roles and delegation of administrative capabilities, beginning with a super administrator and ending with users with DS capabilities. Finally, because the PM displaces AC features that are often implemented in application code to an underlying AC framework, those features can be made less susceptible to bypass and less vulnerable to attack.

The PM is more than just a concept. Through its reference implementation, its features and capabilities have been shown to be viable. The implementation is available from GitHub as an open source distribution to allow wide-spread experimentation and transfer. Example DSs are provided with the distribution, and include messaging, records management, and work flow applications, cut/copy and paste, and several representative office applications.

The PM's architecture and formal model has been adopted by the American National Standards Institute, International Committee for Information Technology Standards (INCITS) as the basis for the Next Generation Access Control standard [1, 2].

6. References

[1] Information technology - Next Generation Access Control - Functional Architecture (NGAC-FA), INCITS 499-2013, American National Standard for Information Technology, American National Standards Institute, March 2013.

[2] Working DRAFT Information technology - Next Generation Access Control – Generic Operations and Data Structures (NGAC-GOADS)), INCITS 499-2013, American National Standard for Information Technology, American National Standards Institute, April 2014.

[3] Mat Blaze, Joan Feigenbaum, and Jack Lacy, Decentralized Trust Management, IEEE Symposium on Security and Privacy, Oakland, CA, USA, pp. 164-173, May 1996.

[4] Mat Blaze, Joan Feigenbaum, and Angelos Keromytis, KeyNote: Trust Management for Public-Key Infrastructures, The 6th International Workshop on Security Protocols, Cambridge, UK, April 1998, in Vol. 1550 of Lecture Notes in Computer Science, Springer-Verlag, pp. 59--63.

[5] Mat Blaze, Joan Feigenbaum, John Ioannidis, and Angelos Keromytis, The KeyNote Trust Management System, Version 2, RFC-2704. IETF, September 1999.

[6] David Ferraiolo, Serban Gavrila, Vincent Hu, and Rick Kuhn, Access Control Policy Management: Composing and combining policies under the policy machine, Symposium on Access Control Models and Technologies (SACMAT), Stockholm, Sweden, pp. 11-20, June 2005.

[7] David Ferraiolo, Vijayalakshmi Atluria, and Serban Gavrila, The Policy Machine: A novel architecture and framework for access control policy specification and enforcement, Journal of Systems Architecture, Volume 57, Issue 4, pp. 412-424, April 2011.

[8] David Ferraiolo, Serban Gavrila, and Wayne Jansen, Enabling an Enterprise-Wide, Data-Centric Operating Environment, IEEE Computer, Volume 46, Issue 4, pp. 10-12, April 2013.

[9] David Ferraiolo, Serban Gavrila, and Wayne Jansen, Policy Machine: Features, Architecture, and Specification, NISTIR 7987, National Institute of Standards and Technology, Gaithersburg, Maryland, 109 pp., May 2014.

[10] Hilary H. Hosmer, The Multipolicy Paradigm for Trusted Systems, Proceedings of the 1992-1993 Workshop on New Security Paradigms, August 1993, Little Compton, RI, USA, pp. 19-32.

[11] The eXtensible Access Control Markup Language (XACML), Version 3.0, OASIS Standard, January 22, 2013, <URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pd