Embedding Security Policies into a Distributed Computing Environment

Udo Halfmann
Winfried E. Kühlhauser

winfried.kuehlhauser@gmd.de
GMD National Research Center For Information Technology, Sankt Augustin, Germany

This paper discusses the implementation of security policies in multipolicy systems. Multipolicy systems are systems supporting a multitude of security policies, each policy governing the applications within its own and precisely defined security domain.

The paper argues that within multipolicy systems, traditional approaches for implementing security policies such as security kernels are both too weak and too strong. In order to support this thesis, we will discuss architectural issues of the implementation of policy separation, policy persistency, total mediation and putting off-the-shelf applications under the control of security policies. Whenever our statements are illustrated by examples, these examples are taken from a case study we implemented for the OSF Distributed Computing Environment.

Keywords: security policy, multipolicy system, information domain, policy domain, custodian, policy separation, policy persistency, reference monitor, micro-kernel, distributed computing environment, DCE.

1 Introduction

In large organizations, many branches and departments have their own unique information security policy that reflects the specific requirements of the individual organizational unit. Mapping the structure of such organizations to a computer system results in a distributed system with a multitude of security domains, where each security domain is governed by its individual and unique security policy.

Multipolicy systems are one of the challenges of the U.S. Department of Defense Technical Architecture Framework for Information Management (TAFIM) [DIS96]. More precisely, the TAFIM reference framework requires that future DoD information systems must

- support information processing under multiple security policies of any complexity or type, including those for sensitive unclassified information and multiple categories of classified information
- support information processing among users with different security attributes employing resources with varying degrees of protection, including users of non-secure resources
- be capable of submitting off-the-shelf application software to a protected environment where a security policy is enforced on the application.

These requirements imply new challenges in many areas of system design. As an example, in order to protect security policies from influencing each other, operating system designers must solve the problem of policy separation within the security kernel itself, a problem that is not covered by the traditional reference monitor approach. Another challenge is the separation
of the security policy domains. Policy domains have to be separated because they constitute protected environments where a single security policy is enforced on any user, application, or resource. A third challenge is policy expressiveness. While access control lists and capabilities are today's common mechanisms to represent access control policies, the scope of security policies as envisioned by the TAFIM reference framework encompasses any type of security policy, including policies for verifying the authenticity of users, systems, or documents, or policies for achieving communication security.

In order to support multipolicy systems and expressive security policies, the custodian model was introduced in [Küh95]. Custodians support the last step of implementing a security policy in which, after risk analysis, security requirements specification, and security model development the policy is represented in terms of the security mechanisms of a system platform and integrated into a security architecture. A custodian is a capsule containing an executable code unit that has been compiled from a policy's program language representation. A custodian constitutes a runtime system for the policy and provides traditional reference monitor properties such as policy tamperproofness and total communication mediation. Additionally, a custodian links the policy to the entities within its associated security domain.

The expressiveness of a security policy implemented in the custodian model is achieved by the policy's algorithmic representation. Access rules (as well as any other rules, such as authenticity rules) are represented by methods of a (C++) class that implements the policy. Methods of this class describe the preconditions for subjects invoking objects within the policy's domain. In a way, custodians are like programs that reside in access control lists, and that are invoked whenever the corresponding object is accessed.

Custodians encapsulate individual security policies; in a multipolicy system, policies thus are separated and protected from each other. In order to implement security domains, a custodian is bound to any entity (such as a user or a resource) within a policy's domain.

Last not least, the custodian model provides a convenient way of abstraction from platform-specific paradigms for naming, separation, communication and persistency, thus allowing the policy developer to focus on the strategic essentials of the policy itself.

This paper focuses on the implementation of the custodian model. It describes the interface between the pair (platform-independent security policy, policy-independent custodian), and how both parts are combined into a platform-specific security policy that can be directly integrated into the framework of a corresponding security architecture. Whenever the concept is described in detail, the custodian implementation within the OSF Distributed Computing Environment (DCE) is used as a reference.

## 2 The Problem

The engineering of a security policy starts with a risk analysis and ends with an executable code unit that is ready for integration into a security architecture (figure 1). Risk analysis identifies threats and security weaknesses of an IT-system or application and results in a set of security requirements such as specified in [Com98]. Those requirements that refer to rules and practices that regulate how sensitive information is managed and protected within a computer
system are the foundation of the (technical) security policy which in the first step is an informal representation of a strategy to meet the security requirements.

Success in achieving a high level of security in a computer system thus depends on the degree of care put into designing, implementing and verifying its security policies. To that end, a security policy is often formalized or semi-formalized in a security model ([BL76, CW87, BN89, San92b, WWK96] and many more). A security model provides a basis for a formal analysis of security properties (such as safety in [HRU76]).

The paradigm shift between formal model and its representation in the security mechanisms of an operating system is often mitigated by representing the model in terms of a specification language such as Z [Dil90] or Skippy [Bry97]. However, the expressiveness of most operating system’s protection mechanisms such as access control lists and capabilities are rather poor and fail to provide an adequate way to implement security requirements such as “every transaction must be logged in a way so that the log can be used as evidence before court”.

Security policies in the custodian model are implemented by programs. As a consequence, the next step transforms a policy specification into its representation in a programming language. Depending on the abstraction level of the specification, a programming language representation can often be generated by automatic tools. Figure 2 shows the major parts of a C++-class implementing a simplified Chinese Wall security policy using the lattice [Den76, San92a]. The class was generated automatically by [KP99] from a Skippy specification.

Any such class consists of two major sections, a variables section and an operations section. The variables section describes the state of a policy; in our example, the policy state consists of a lattice describing mandatory information flow, a classification function that assigns lattice labels to each entity within the policy’s domain, and an access matrix describing discretionary access rights. The operations section contains the rules that control entity interactions. The example in figure 2 is an access control policy defining two rules for subjects invoking read and write operations on objects. This section may also contain operations to modify the policy state, e.g.


```cpp
class Policy: public ChineseWall, public Custodian {
  /**< policy variables: lattice, labels, entity classifications, and discretionary access control */
  bool lattice [SysHigh][SysHigh];
  enum label_t {SysLow, bank1, bank2, oil1, oil2,
               bank1oil1, bank1oil2, bank2oil1, bank2oil2, SysHigh};
  label_t classification [MaxEntities];
  enum rights {none=0, read=1, write=2};
  rights_t access_matrix [MaxEntities][MaxEntities];

  /**< policy operations: read and write rules */
  bool read (int s, int o) {
    return lattice[classification[o]][classification[s]] && (read & access_matrix[s][o])
  }
  bool write (int s, int o) {
    return lattice[classification[s]][classification[o]] && (write & access_matrix[s][o])
  }
};
```

Figure 2: An example security policy represented as a C++-class
to change a discretionary right in the access matrix or the classification of an entity. Optionally,any such C++-class representation of a security policy may contain additional variables (such as encryption keys) and additional mechanisms (such as encryption algorithms).

A C++-class representing a security is still free from platform-specific paradigms or mechanisms (see figure 2). It thus can be easily re-used on different platforms. However, security policies do have strong requirements to their runtime environment. As an example, the total communication mediation property of the reference monitor—a widely known implementation concept for access control policies—requires that a policy rule is applied to any communication within a policy’s domain and thus establishes a strong affiliation between policy and a platform’s communication system. As another example, the statefulness of a policy requires that the policy state is maintained in persistent memory.

Our approach to integrate a security policy in a specific environment is to embed a policy in a policy-independent but platform-specific runtime system. Thus any executable security policy is a pair (platform-independent security policy, policy-independent runtime system), a pair we called custodian in [Küh95].

While the first part of this pair is a security engineering topic (figure 1), the focus of this paper is the second part. We will describe the general software architecture of a security policy runtime system for multipolicy systems. We will discuss the implementation of policy tamperproofness, total communication mediation, policy persistency and the linking of a policy to its domain. Whenever we illustrate our ideas by examples, these examples are taken from the custodian implementation for the OSF Distributed Computing Environment.
3 Software Architecture

The software architecture of a custodian consists of five components. The central component is the platform-independent security policy which is wrapped in a runtime system consisting of four components that implement policy tamperproofness, communication mediation, persistency and domain linking.

![Custodian Architecture](image)

**Figure 3: Custodian Architecture**

In client/server terms,

- the **total mediation component** interfaces the security policy to the communication system of the underlying system platform. This component is called by hooks within the communication system and acts as a client seeking policy decisions from the security policy.
- the **domain linking component** has two closely related tasks. Firstly, it binds the security policy to each entity within the policy's domain. Secondly, it acts as a name server for the total mediation component, mapping real-world entity names (such as Unix pathnames or DCE unique identifiers) to policy-level entity names (such as the integers $1..\text{MaxEntities}$ in figure 2).
- the **persistency component** is the interface to the persistent memory system of the underlying system platform (in general the file system). It acts as a persistency server for the security policy as well as for the domain linking component.
- the **tamperproofness component** protects the security policy by using the address space separation scheme of the underlying system platform. It is executed during custodian initialization and has no further interface to other components.

These four components together with the security policy constitute a custodian. Within this section we describe each of these components and their interfaces to the security policy. In contrast to the inner interfaces, the interfaces to the communication system, the address space scheme and the persistent memory system are highly dependent on a specific system platform.
3.1 Total Mediation

One of the basic reference monitor requirements for implementing access control policies is that a security policy must be in a position to mediate any communication between the entities in its domain. Considering multipolicy systems, this requirement has two different aspects.

Firstly, we must guarantee that communication control is absolute. This requires that a security policy is able to modify or suppress any information flow within its domain. On the DCE platform, clients and servers in general communicate by DCE’s remote procedure call mechanism. Our total mediation implementation for the DCE assumes that this is the only way for entities to communicate. This assumption is not valid in general; any DCE client or server may still directly use the communication mechanisms of the underlying operating system, such as sockets, Unix pipes, or even files. However, any client/server communication that circumvents the DCE RPCs requires the active cooperation of the server, and our assumption is that any well-written and trustworthy server will refuse this. Assuming trustworthiness of servers in the DCE is in many cases inevitable anyhow, because servers have a high degree of control on their objects. They control their objects with respect to availability, and, if clients do not use encryption, servers also control confidentiality and integrity. However, whenever a mandatory access control policy has to be enforced on an untrusted server, additional server validations (such as static code analysis) are obligatory.

Within the context of the DCE, four general possibilities for mediating the RPC communication exist. They are

- locating the mediation point within the communication system of the operating system
- locating the mediation point within the DCE runtime system
- locating the mediation point within the server or client stubs
- locating the mediation point within the server or client code.

Criteria for choosing one of these alternatives are the resulting size of the trusted computing base (TCB), the availability of server and client source code, the operating system’s API (some operating systems allow the integration of user code) and the convenience of the application programmer. A more detailed discussion can be found in [HKK93, SHK+94, Min95, Lux95, OFSS96]. As one example, figure 4 shows a client/server call where the mediation point is located within the socket system of the underlying operating system; the custodian is implemented as a regular DCE server (see below).

In client/server terms, the total mediation component acts as a server serving policy decisions to all mediation points. It maps mediation points to policy rules, uses the services of the domain linking component in order to map real-world entity names to policy entity names, calls the policy and enforces the policy decision by granting or denying the communication.

In multipolicy systems, different entities may belong to different policy domains. As a consequence, the second aspect of total communication mediation is that for each entity interaction it must be decided which security policy to apply [Kühl98]. One approach is that mediation points broadcast a request for policy decisions to every security policy within a system. However, this solution has several drawbacks. Firstly, it interferes with policy autonomy; an entity interaction that completely lies within the domain of a single policy must not be influenced by any other
policy in the system. Secondly, if an entity interaction involves more than a single policy, the relationship between the involved policies (such as priorities or algebraic expressions of boolean access decisions) is not clearly defined. [Küh98] describes a method for mediation in multipolicy systems; the method uses a two-level scheme where all mediation points direct their policy requests to a commonly trusted metapolicy that implements a strategy for combining multiple policies. Such metapolicies are implemented using the same custodian paradigm as regular policies to implement total mediation, tamperproofness, domain linking, and persistency.

### 3.2 Tamperproofness

Tamperproofness of a security policy encompasses the policy’s active as well as its passive phases. Like any regular program, an active security policy has threads and memory segments, and it frequently uses processors and changes its state. A passive policy is frozen and committed to persistent memory.

Tamperproofness of a passive security policy is implemented by the persistency component (see below). The tamperproofness component implements policy tamperproofness at runtime, making straightforward use of the address space separation scheme that the underlying system platform provides for separating regular user applications. On a Unix platform, a custodian would be implemented as a Unix user process. Within the DCE, a security policy is implemented as a regular DCE server, thus profiting from DCE’s platform-independent separation scheme. Because address space separation is established once whenever a custodian enters an active phase, no interface between the security policy and the tamperproofness component is required.

### 3.3 Domain Linking

The domain linking component implements a mapping between the real world within a computer system and the abstract world within a security policy. Domain linking is basically a name service, mapping real-world entity names to policy-world entity names and real-world entity operations to policy-world security rules. Additionally the name server has a second purpose: the set of all entities within the domain linking component constitutes the policy domain.

---

1 In single-level storage systems, the need for considering these states differently becomes obsolete.
Real-world entities and a policy-world entities are associated whenever an application (a set of entities) is put under the control of a policy. As applications and policies in general are bound together dynamically [DIS96], the domain of a policy may change over time. Consequently, binding operations as well as application operations that create or destroy entities are mapped to corresponding name server operations.

3.4 Persistency

Security policies in general exist for a long period of time (years). Policy states thus must survive normal system shutdowns as well as occasional system crashes. Persistency mechanisms implement this property by saving and recovering a custodian’s state to persistent memory.

The major parts of a custodian’s state are the policy-dependent variables (access matrix, lattice, entity classification), the domain and name server tables, and the custodian code. As mentioned in section 3.2, policy tamperproofness encompasses the policy’s persistent state and thus implies that a system’s persistent memory implementation is a major part of the trusted computing base.

Regular file systems do not really meet the requirements pointed out in the reference monitor principles; file systems rarely are separated from the rest of the operating system, they never are small and well-structured so that they can be formally analyzed, and they very often have distributed components (NFS, AFS). Obviously, $\mu$-kernel architectures that provide a simple persistent memory service directly on top of the $\mu$-kernel are considerably better suited to meet the requirements for implementing security policies.

4 Implementation

This sections describes a custodian implementation for the OSF Distributed Computing Environment. The work was part of a master thesis and was done in order to get insight into the practical aspects of our approach and to demonstrate that state-of-the-art software engineering methods are applicable. As a consequence, several substantial decisions discussed in the previous section were taken in a pragmatic way, resulting sometimes in an insecure implementation and a pretty large TCB.

OSF DCE is a middleware software system to support distributed applications in heterogeneous distributed systems. To this end, DCE provides a client/server paradigm as well as RPC communication and distributed services such as directory services and time services.

Figure 5 shows the detailed C++ class hierarchy of a custodian. The class hierarchy consists of three levels. The top (meta-)level contains only abstract classes in the sense that all methods are defined with their parameters. Each method defined here is implemented in a derived class on a lower level. At the meta-level the functionality of each component is described by a collection of class headers defining the interfaces of the architecture components (cf. figure 3).

At the platform level, each individual system platform (such as DCE, Sun Solaris, or Windows-NT) is represented by a set of derived classes that are platform-specific implementations of the abstract classes of the meta-level. To preserve the classes’ interface no additional public methods exist on this level.
At the policy level the abstract class representing a security policy is implemented. The policy uses the other component’s functionality by creating instances from the platform level classes. All classes on the policy level are linked together by an ordinary C++ compiler; the result is an executable custodian.

4.1 Total Mediation

The starting point for any total mediation implementation is the underlying communication system. As communication paradigms are platform-specific, we have to deal with low-level communication mechanisms (like UNIX pipes or message passing) as well as with high-level RPC communication systems. While our basic ideas apply to low-level mechanisms as well, we here focus on RPCs.

In a RPC client/server environment, the set of procedure headers exported by a server is called an interface. Server interfaces are a major part of any DCE application. All server interfaces of an application (consisting of many clients and servers) constitute the application interface. When application operations are mapped to policy rules (i.e. the policy interface), server interfaces have to be mapped to policy interfaces. We consider three alternatives to this mapping:

1. The first approach is a direct 1:1-mapping of application interfaces to a policy interface where every application-specific operation is mapped to a corresponding policy rule. The advantage is that the policy can fully exploit the application interface semantics of the

Figure 5: Class Hierarchy
application operations. A disadvantage is that the policy must be adapted to each modification of the application interface, which limits the policy's generality and reusability.

2. In the second approach there is exactly one generic policy operation which is called on every application RPC. An additional API (the methods of class MetaParams in figure 5) allows the policy to optionally retrieve the name of the application operation and its parameters. While providing the same expressiveness, the advantage of this approach is the policy’s higher level of independence from the interface of the application; however, we have to pay for it with a bigger overhead when operation names or parameters are needed to compute a policy decision.

3. The third approach uses a separate mapping function in order to associate policy rules to the operations of the application. This approach is often used in traditional security policies that define rules only for read and write operations. Here, the mapping of application operations to policy rules reduces the semantics of application operations to a simple read or write semantics. In this approach, we trade expressiveness for generality and reusability.

In many distributed computing environments (such as DCE or CORBA), server interfaces are described by an interface definition language (IDL). IDLs allow for the automatic generation of header files, client stubs and server stubs. In order to reduce the complexity of our prototype implementation we sketched a simple, easy-to-process IDL, and an additional mapping definition language (MDL) for the third approach mentioned above. In order to support the first alternative, the abstract class from which the policy class is derived (e.g. class MetaChineseWall in figure 5) is generated automatically from the IDL interface descriptions of the application. In the second alternative the policy interface is not application-dependent, thus a single abstract class for all policies is sufficient. For the third alternative, the abstract class is generated from an MDL description.

Now let us consider the class definition of a policy. Any RPC between a client and a server consists of two communication operations, a request and a reply. This scheme maps to operations int generic_request (client, server, params) and int generic_reply (client, server, params) which directly support the second approach. With respect to the other approaches additional request and reply operations with corresponding parameters are available for each individual policy operation. In order to put a policy into the position to change parameters of an application operation, all parameters are inout types. All policy operations return a boolean value which indicates whether communication may proceed.

The hook (mediation point) for communication control is located in the stubs (see the discussion in section 3.1). While this decision results in a large TCB (stubs share a client’s or server’s address space and thus hooks in stubs can be manipulated by them), we nevertheless achieve the functional properties we need to reach our goals.

Communication between hooks and custodian servers is achieved by an ordinary, application independent RPC interface. While this interface is basically very general, it causes quite an overhead in cases where generality is not really needed. As an example, many types of policies (such access control policies) have no need for changing application operation parameters. In order to reduce the cost of these communications, the tools that generate the interfaces have options by which the interfaces can be tailored. Additionally, caching within mediation points reduces policy communication; caching works well, because policy rules do not change very often.
4.2 Domain Linking

The binding between application entities and their security policy is implemented by two methods of the **MetaCustodian**-class. The **link** operation is called when a new entity in the policy's domain is created and the **unlink** operation is called when an entity is deleted. These operations are part of both, the custodian’s RPC interface and the policy’s interface.

Called by appropriately placed hooks, the operations first insert respectively delete the entity’s real word address into the custodian’s name mapping table. This table translates the entities’ platform-specific names into policy-level entity names (implemented by class **MetaEntity**). After that, the policy operation is called with the internal policy name of the entity as parameter.

The class **MetaEntity** essentially contains an integer field which identifies the entity and additionally is often used by the policy to index tables or lists. The value is made unique in the time by a version field, which is incremented whenever an integer identifier is re-used.

4.3 Tamperproofness

Tamperproofness of a security policy is a side effect of implementing every custodian as an individual DCE server. Any DCE server runs in a separate address space that usually is provided by the process separation scheme of the underlying system platform. Of course, this decision again explodes the size of the TCB.

Any custodian has two parts. One part is encapsulated within a DCE server and consists of the security policy, the tamperproofness component and the persistency component. The second part consists of fragments that are distributed among the communication and entity management system of the DCE. These fragments trigger the custodian on communication operations or entity creation and deletion by using regular RPCs. Thus all DCE components that contain such fragments obviously become part of the TCB.

While the DCE server paradigm protects a policy during the custodian’s runtime, in passive phase its state is saved in the file system and protected by the regular UNIX file system security mechanisms. This decision makes the complete Unix file system a part of the TCB.

4.4 Persistency

This component is used to write the policy state to persistent memory. It implements a mechanism to store the data of C++ classes. For this purpose the abstract class **MetaObject** is defined, from which all classes must be derived that want to use the persistency server. The class contains abstract methods which store the class’s data into a regular file. The methods of all derived classes are called by the persistency component automatically to save the custodians entire state. The state is restored whenever a custodian reenters its active phase.

4.5 Policy Modules

When implementing security policies it is a general observation that several security mechanisms (such as authentication protocols, access matrices, auditing, or encryption algorithms) are used
in many different policies. We thus collected many of these mechanisms in a class library (classes LibStd, LibACM, ACM) so that they can be used in every policy.

Security mechanisms can be implemented both in a platform-dependent as well as in a platform-independent manner. As an example, an access matrix implemented on top of a Unix platform can either use Unix ACLs or, if the expressiveness of the owner/group/others rwx scheme appears to be too poor for a given policy, a more elaborate representation based e.g. on sparsely populated matrices can be selected.

These options suggest the integration of the class library into the three-level hierarchy of figure 5. The meta-level of the hierarchy provides the common interface, while the platform level provides implementation versions that are tuned to the specific needs of a policy.

5 Conclusions

As a consequence of our argumentation it becomes obvious that within a multipolicy system, implementing a security kernel (defined as that part of a trusted computing base that implements the reference monitor concept) by a separated and monolithic system component operating within a single address space counteracts the three reference monitor principles. We discussed a software architecture that implements security policies, policy separation, policy persistency, total mediation of distributed applications and multiple distributed policy domains by a collection of isolated components that interact by small interfaces.

Since the early 80's the reference monitor principles influence the architecture of secure operating systems. These principles provide strict rules for arranging the components of a security kernel. The first reference monitor principle implies that different security policies must be separated from each other. In order to enforce policy separation even in presence of malicious (or erroneous) security policies, separation itself must be implemented in a protected system area that is separated from the policies. The third reference monitor principle implies that separation implementation must be small, well-structured and protected. Both principles together thus provide strong arguments (necessary and sufficient conditions) for a $\mu$-kernel based system architecture where separation is enforced within the $\mu$-kernel.

The second reference monitor principle implies that any application-level communication is mediated by a security policy. A non-monolithic security kernel additionally implies that system-level IPC becomes crucial for guaranteeing security kernel correctness. Again, taking into account the third reference monitor principle, strong arguments exist to locate communication control within a $\mu$-kernel.

In order to get some insight into the practical aspects of our approach, we made an implementation for the OSF Distributed Computing Environment. While quite obviously several pragmatic design decisions of this implementation resulted in a pretty large TCB, we nevertheless were able to outline practical issues and provide a detailed software architecture that identified the major components of a distributed multipolicy security kernel. Additionally, it became quite clear why a conventional implementation on top of an operating system is highly insecure and also causes the TCB size to explode.

Future work will aim at improving our level of trust in the implementation by reducing the
TCB size and at improving the efficiency and security of the implementation by a well-balanced mapping of security kernel components to $\mu$-kernel-based operating system architectures.

References


[Com98] NIST (USA), CSE (Canada), BSI (Germany), NNCSA (The Netherlands), NSA (USA), CESG (UK), SCSSI (France). Common Criteria for Information Technology Security Evaluation, Version 2.0, May 1998.


