The following paper was originally published in the

Proceedings of the 8th USENIX Security Symposium

Washington, D.C., USA, August 23–26, 1999

SCALABLE ACCESS CONTROL FOR DISTRIBUTED OBJECT SYSTEMS

Daniel F. Sterne, Gregg W. Tally, C. Durward McDonell, David L. Sherman,
David L. Sames, Pierre X. Pasturel, and E. John Sebes

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Scalable Access Control for Distributed Object Systems

Daniel F. Sterne (dan_sterne@nai.com)
Gregg W. Tally (gregg_tally@nai.com)
C. Durward McDonell (durward_mcdonell@nai.com)
David L. Sherman (david_sherman@nai.com)
David L. Sames (david_sames@nai.com)
Pierre X. Pasturel (pierre_pasturel@nai.com)
NAI Labs, Network Associates, Inc.

E. John Sebes (ejs@securify.com)
Kroll-O'Gara Information Security Group

This research was funded by DARPA under contract F30602-97-C-0268.
Approved for Public Release, Distribution Unlimited.

Abstract

A key obstacle to the widespread use of distributed object oriented systems is the lack of scalable access control mechanisms. It is often necessary to control access to individual objects and methods. In large systems, however, these can be so numerous that the resulting proliferation of access control information becomes overwhelming. We describe Object Oriented Domain and Type Enforcement (OO-DTE), a technology for organizing, specifying, and enforcing access control that has been prototyped and integrated with commercial ORBs and SSL. OO-DTE provides fine-grained control and scalability via a compilable symbolic policy language. We discuss our experience building and using OO-DTE and compare OO-DTE with the access control terminology, concepts, and requirements described in CORBA Security.

1. Introduction

Although distributed object technology has matured greatly during recent years, the lack of practical, integrated security mechanisms remains an obstacle to its use in many application domains. CORBA[1] and Java Remote Method Invocation (RMI)[2] are two of the more popular distributed object technologies. Version 1.0 of the CORBA Security specification (CORBASec) [CORB] was initially released in January 1996.

[1] Common Object Request Broker Architecture from the Object Management Group (OMG)
[2] Java is a product of Sun Microsystems.

However, commercial product releases that fully comply with its Level 2 (general-purpose) requirements are just beginning to emerge. Moreover, even Level 2-compliant products may prove inadequate because Level 2 requirements in some functional areas are neither complete nor sufficiently stringent. In particular, CORBA Security does not establish requirements for security management mechanisms. These mechanisms can have a major impact on the usability and scalability of the implementation. Java security, as represented by Sun's Java 2 and the recently announced Java Authentication and Activation Service (JAAS), is still undergoing rapid evolution. Moreover, although Sun has released plans and a draft specification for the RMI Security Extension [RMISEC], the facilities currently provided for controlling RMI-based access by clients to remote Java servers are very limited.

Providing practical access control mechanisms for distributed objects - whether based on CORBA or Java RMI - is a challenge because the necessary characteristics for such mechanisms are not well understood. One particular problem stems from the common requirement to control access to individual objects and methods, even in systems where the number of these abstractions is enormous. For example, consider the CORBA naming service, which allows CORBA objects to be bound to portions of CORBA's hierarchical name space and provides queries about name bindings. It would be unwise to give every user complete freedom to use the naming service, i.e., unrestricted ability to *bind* any object to any part of the name space, or *unbind* any name-to-object binding.

Mechanisms are needed to allow a user to invoke these naming service methods, but only on appropriate objects. Unfortunately, attempting to enumerate separate access control attributes for every method of every object in a large Object Oriented (OO) system causes an overwhelming proliferation of access control information that results in a loss of understandability, manageability, and ultimately, security. This problem is worsened when Access Control Lists (ACL) are used as security attributes [OSG] [DENG] [HU] because each ACL can contain many entries, including some that appear contradictory. Moreover, the combined effect of ACLs can only be understood by applying complex precedence and ordering rules [BALD] [DOWN].

This paper describes the results of a DARPA-funded research project to develop access control technology for distributed objects. By access control technology, we mean mechanisms for 1) specifying the sets of objects and methods that users and automated processes may access, and 2) protecting objects and methods from unauthorized access. The technology we have developed is called Object Oriented Domain and Type Enforcement (OO-DTE). OO-DTE is an outgrowth of our earlier research into access controls for secure operating system kernels [BADG] [WALK] [SHER].

OO-DTE was developed with the following goals:

- Object Oriented - It should support OO abstractions and take advantage of OO relationships, especially interface inheritance.
- Scalability and Manageability - It must be suitable for controlling access in large distributed applications systems comprising thousands of processes, objects, and methods.
- Fine-grained Control - It must allow access to be controlled at the level of individual objects and individual methods.
- Role-based Access Control (RBAC) - It should support access rules organized according to user roles (job titles or functional responsibilities) rather than user identities or security clearances.
- Compatibility with Commercial Products – It should be designed as a plug-in module that can be linked into commercial object request brokers (ORBs) and distributed object infrastructure components.
- Transparency - It should support *security unaware* applications, i.e., applications that do not explicitly invoke access control or other security services.

This paper is organized as follows: Section 2 provides background, terminology, and motivation. Section 3 describes DTEL++, the compilable language used in OO-DTE to specify access control policies. Section 4 describes the implementation, features, and status of our OO-DTE prototypes. Section 5 compares OO-DTE to CORBA Security concepts and terminology. Observations on our experience with OO-DTE, including preliminary performance data, are discussed in Section 6. Section 7 provides a summary.

2. Background

OO-DTE extends Domain and Type Enforcement (DTE) concepts to distributed object systems. DTE is a set of access control mechanisms for UNIX kernels [BADG] [WALK] [SHER]. DTE was, in turn, based on Type Enforcement, as proposed originally by Boebert and Kain [BOEB] in 1985. For historical reasons, DTE and OO-DTE have continued to use terminology that originated from Type Enforcement even though those terms are now overloaded with additional meanings.

On a DTE operating system, a security attribute called a *type* is associated with every file, directory, device, and IP packet. Types are assigned to these system resources according to a site-specific policy. Types are typically assigned to indicate the kind of information contained, its sensitivity, integrity, or origin. A security attribute called a *domain* is associated with every process. Domains are assigned to processes to indicate the kind of computing tasks they are intended to perform and to prevent unnecessary access to system resources. Each domain is defined as a collection of access rights. Each right is expressed as the ability to access information of a specified type in one or more access modes, e.g., read, write, execute, send, receive, etc.

One of the innovations of DTE was the use of a compilable high-level language to define types and domains. This language is called the DTE Language or *DTEL*. DTEL was also used to assign types to the file hierarchy via a set of general rules with exceptions. This allowed an entire file system, potentially consisting of millions of files, to be "labeled" in a concise and understandable manner by means of a few DTEL statements. This contrasts with ordinary UNIX systems that require an administrator to examine the permission bits on vast numbers of files and directories *individually* in order to infer how files of different sensitivities are arranged and distributed throughout the file hierarchy.

OO-DTE is an attempt to extend DTE notions to distributed object-oriented systems. We have concentrated initially on developing OO-DTE for CORBA-based systems but have plans to adapt OO-DTE for Java RMI. OO-DTE includes an extended policy language called DTEL++ that provides constructs for assigning types to methods. DTEL++ also includes rules for propagating default type assignments from modules and interfaces to enclosed methods[3], and for propagating type assignments to inherited methods. DTEL++ defines two new access modes for methods: *invoke* (needed by clients) and *implement* (needed by servers).

The first OO-DTE prototype ran on a DTE kernel and exploited the DTE kernel's access control facilities for interprocess communications [SHER]. That prototype, sometimes referred to as *Kernel Level OO-DTE*, involved using and modifying the Inter Language Unification (ILU) ORB[4]. These modifications were needed to make ILU use the DTE kernel's extended *send* and *receive* system calls, which transmit type attributes with messages and prevent senders and receivers from accessing unauthorized information types.

Although the DTE kernel provides important security advantages, there is little interest outside the research community in non-standard kernels. To broaden the potential audience for our research we have subsequently developed newer prototypes called *Above Kernel OO-DTE*. Above Kernel OO-DTE was specifically designed to run on mainstream commercial operating systems and is the primary focus of this paper.

2.1. Domains, Roles, and Role Authorization

OO-DTE has been designed to support role-based security policies, that is, policies that grant users access rights according to their assigned roles (duties) within an organization [NICO]. In our earlier DTE kernel operating systems, each role was represented as a collection of domains. This allowed each task initiated by a user to run as a separate process in a "small domain" providing only the minimum set of access rights needed to accomplish that task. This approach was developed in accordance with the principle of least privilege [SALT]. OO-DTE, however, is designed to run on mainstream operating systems whose kernels lack comparable process spawning and access control facilities. Consequently, in OO-DTE, each role is represented by a single domain; hence, the terms *role and domain have become virtually synonymous*.

In OO-DTE, a user may be authorized for multiple roles, but each connection associated with a user process is bound to a single role (domain). OO-DTE roles can be constructed hierarchically by defining domains in terms of other domains, in the manner of Baldwin's Named Protection Domains [BALD]. OO-DTE does not provide facilities for dynamic or static separation of duty; unlike some practitioners [KUHN], we regard these as distinct from the fundamental aspects of role-based access control.

Roles, domains, and types provide several levels of indirection in the authorization policy. As discussed later, users hold X.509 certificates that contain an OO-DTE domain as a privilege attribute. The policy defines domains in terms of their access rights to types. The policy also assigns types to methods. Through these levels of indirection, a user's authority to invoke a method can be determined. Figure 1 shows the relationships between users, domains, types, and methods:

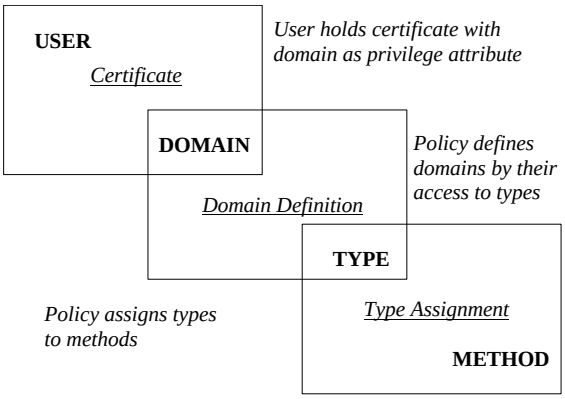

Figure 1 - Users, Roles, and Authorization

3. DTEL++ Policy Language

DTEL++ is the policy language used to specify OO-DTE security policies [TALL]. It is used to declare types, to assign a type to each method that might be invoked, and to define domains in terms of the types each domain can invoke or implement. We will motivate the discussion of the policy language by illustrating how it can be used to control a simple application system.

[3] Note that CORBA uses the terms "interface" and "operation" in place of "class" and "method". We use both sets of terms interchangeably.

[4] ILU was developed by Xerox Corporation, Palo Alto Research Center.

3.1. Sample Application

Suppose we are writing an application to manage the books in a library. The primary objects in our application are librarians, patrons, and books. One can enter, remove, and find books in the card catalog, reserve books, and check books in and out. The Client-server interface for this application, as specified in CORBA's Interface Definition Language (IDL), might look like this:

```
module Library {
  struct BookDescription {...};
  interface Patron {...};
  interface PatronDatabase {...};

  interface Book {
   readonly attribute BookDescription desc;
   void checkOut (in Patron patron);
   void checkIn ();
   long numberAvailable();
   long numberReservations();
   void reserve (in Patron patron);
   };

  interface BookDatabase {
    Book newBook (in BookDescription desc,
                  in long copies);
    void removeBook (in Book book);
    Book findByTitle   (in string title);
    Book findByAuthor  (in string author);
    Book findBySubject (in string subject);
  };
};
```

In our policy, we want to allow both librarians and patrons to find books in the card catalog and to reserve books. Additionally, we want to allow only librarians to check books in and out, to enter new books into the catalog, and to remove books from the catalog.

3.2. Type Definitions and Simple Type Assignments

Every DTEL++ policy must declare types. These types are then used to label interface methods so that access to the methods can be controlled. In our policy we declare two types, `safe_t` and `restricted_t`[5] with the OO_type statement:

```
OO_type   safe_t, restricted_t;
```

Methods that both patrons and librarians are allowed to invoke will be assigned the `safe_t` type, and methods that only librarians are allowed to invoke will be assigned `restricted_t`.

Each method in each interface must be assigned a type. This can happen in one of several ways. The most straightforward way is with an explicit assign statement for the method. Since patrons can safely be allowed to look up books in the catalog, the type `safe_t` is assigned to the method `findByTitle()` by the assign statement:

```
assign   safe_t   findByTitle;
```

A second way of assigning types to methods is to assign a default type to the methods within an interface or module, as in:

```
assign   restricted_t _DEFAULT;
```

Each module and interface may be assigned a default type. By declaring a default type of `restricted_t`, any method that does not receive a type via an explicit assign statement, as shown previously, will receive the type `restricted_t`. This allows us to construct our policy so that patrons can invoke only those methods that have been explicitly granted to them.

Other ways of implicitly assigning types will be discussed in Section 3.5.

3.3. Domain Definitions

A DTEL++ policy must define the permissions granted to each domain. For each type declared in the policy, a given domain can have `invoke` permission, `implement` permission, neither permission, or both permissions. Only those permissions explicitly stated in the domain definition are granted to the domain. In our example application, we can allow patrons to perform safe operations, and no others, with the domain declaration:

```
domain patron_d = (invoke->safe_t);
```

We permit librarians to perform both safe and restricted operations with:

```
domain librarian_d =(invoke->safe_t,
                                restricted_t);
```

Domains can be concatenated. Since the `librarian_d` domain has a superset of the privileges of the `patron_d` domain, it could have been defined as:

```
domain librarian_d = patron_d,
                     (invoke->restricted_t);
```

While the impact upon our simple example is minimal, had the definition of the `patron_d` domain been substantially more complex, the above construction of the `librarian_d` domain would still allow the policy to show very clearly the relationship of the two domains. This allows one to express larger policies in a

[5] We have adopted the convention that the suffixes "_t" and "_d" denote type and domain identifiers

more concise, and more understandable way, for example, policies in which roles are defined hierarchically.

The software that maintains the library database must implement all of the methods and requires the following domain definition:

```
domain server_d = (implement->safe_t,
                           restricted_t);
```

For an example of both `invoke` and `implement` permission, consider augmenting the `server_d` domain. We may want to give it the `invoke` privilege for the `safe_t` type if it has to invoke some of the listed methods internally to do its job. We would re-define the server domain statement as this:

```
domain server_d = (invoke->safe_t),
       (implement->safe_t, restricted_t);
```

3.4. Sample Policy

Here is the DTEL++ policy for our library system:

```
OO_type safe_t, restricted_t;

module Library {

  assign restricted_t DEFAULT;

  interface Book {
    assign safe_t { _get_desc,
                    numberAvailable,
                    numberReservations, reserve
                    };
  };

  interface BookDatabase {
    assign safe_t { findByTitle,
                    findByAuthor,
                    findBySubject };
  };
};

domain patron_d    = (invoke->safe_t);
domain librarian_d = (invoke->safe_t,
                      restricted_t);
domain server_d    = (implement->safe_t,
                       restricted_t);
```

The first line declares the two types that we are using. Next, we open the `Library` module and assign `restricted_t` as the default type for all methods in the module. Then we open the `Book` and `BookDatabase` interfaces and explicitly assign the `safe_t` type to methods that may be invoked by patrons. Finally, we define our domains: applications run by patrons will run in the `patron_d` domain and be allowed to invoke only `safe_t` methods; librarian applications will run in the `librarian_d` domain and be allowed to invoke `safe_t` and `restricted_t` methods; and any servers will run in the `server_d` domain and be allowed to implement both `safe_t` and `restricted_t` methods.

When an assign statement for a default type appears within the scope of an interface statement, the default type applies to the methods of that interface. When such an assign statement appears outside the scope of an interface statement, but within the scope of the module statement, the default type is accepted as the default type by all interfaces within the module that do not contain their own default assign statement to override it.

In our example, we assigned a default type of `restricted_t` for the entire Library module. All interfaces within the module will have `restricted_t` as their default type since none of our interfaces contains an assign statement to reset the default type within the interface. All methods in the `Patron` and `PatronDatabase` interfaces, which may be numerous, need not be mentioned in the policy; they will receive the `restricted_t` type by default. In the `Book` interface we assigned the type `safe_t` to some of the methods, but since we did not explicitly assign types for the `checkIn()` and `checkOut()` methods, they both receive the interface default of `restricted_t`.

3.5. More Complex Type Assignments

Inheritance

By default, DTEL++ method types are assigned recursively from base interfaces to derived interfaces. For example, assume an interface `ChildrensBook` that derives from the `Book` interface described above. The `ChildrensBook::checkOut()` method would automatically receive the same type, `restricted_t`, that was assigned to the `checkOut()` method in the `Book` interface. It is also possible to explicitly assign a type to the `ChildrensBook::checkOut()` method so that the inherited type is overridden.

The inheritance of type assignments in DTEL++ can be fine-tuned through the use of the **`-i`**, **`-l`**, and **`-f`** flags. These flags appear immediately after the keyword `assign` and modify the effect of the type assignments on inherited methods or derived interfaces. These are discussed further in [TALL].

Per Object Access Control

DTEL++ allows different objects having the same interface to be assigned different access control characteristics, but requires that such objects be bound to names like those defined by the CORBA Naming Service. Objects with different names can then be treated differently.

For example, there may be some books in a library that no one may check out, perhaps because they are antiques. Instead of having to define a new `AntiqueBook` interface, we can leave the IDL for the application unchanged and add the following lines to our DTEL++ policy:

```
// No one has permission to invoke null_t
OO_type null_t;
...
module Library {
  ...
  template AntiqueBook : interface Book {
    assign null_t checkOut;
  };
  assign AntiqueBook /Books/Antique/;
  ...
};
```

The collection of type assignments for a given interface is called its *default type template*. This includes both explicit and implicit type assignments received through inheritance relationships or default types. An object that is not named has the *default type template* applied to it. The example above defines a *named type template* `AntiqueBook`. The *named type template* applies only to objects with names that fall under the namespace assigned to the type template (/Books/Antique/).

Object names are arranged in a directory-like structure, and are given to the objects by security aware object factories. In our example, an antique book should be bound to a name such as "/Books/Antique/1003" (for book #1003 in the catalog, for example), while a regular book should be bound to a name such as "/Books/1351" (or possibly no name at all). The following DTEL++ line says that any object with a name that starts with "/Books/Antique/" should have its methods typed as in the `AntiqueBook` template, rather than the standard (unnamed) `Book` template.

```
assign AntiqueBook /Books/Antique/;
```

4. Implementation

Several OO-DTE prototypes have been built. This paper focuses on two more recent prototypes that have been built as plug-in modules for two commercial ORBS running on mainstream operating systems: Orbix (Iona Technologies) on Solaris and Visibroker (Inprise) on Windows NT.

4.1. DTEL++ Compiler

The DTEL++ compiler uses DTEL++ policy files and associated CORBA IDL files as input to generate data files used by an OO-DTE ORB at runtime. The compiler output consists of 1) tables that bind types to methods, and 2) domain definitions that are described in terms of the types each domain can invoke or implement. The compiler compares the module, interface, and method identifiers in a DTEL++ policy with their counterparts in the IDL files and reports inconsistencies. If a method, interface, or module appears in the DTEL++, but not in the IDL, it is reported as an error. However, a method may legally appear in IDL and not in the DTEL++. In this situation, the compiler uses the type assignments provided by inheritance and default types to infer the type of the method.

4.2. Run-time Architecture

Like other ORBs, both Orbix and Visibroker provide a "plug-in" interface for extending the ORB's functionality. The CORBA specification calls these plug-ins *interceptors*. Although a standard interceptor interface is under development by the OMG, current interceptors for ORBs are vendor-specific.

The primary runtime components of our Orbix and Visibroker OO-DTE prototypes are interceptor-like plug-in components that reside in both the client and server ORBS. The behavior of these plug-in components is driven by the data files produced by compiling DTEL++ policies. The plug-in in the server's ORB protects the server. For each operation request it receives, it determines whether the requesting client is authorized to *invoke* the requested operation; if not, it rejects the request and sends a CORBA:NO_PERMISSION exception to the client. The plug-in in the client's ORB protects the client. For each operation, the client ORB determines whether the server is authorized to *implement* the operation (i.e., provide the service); if not, the ORB rejects the request, thereby preventing it from being sent to the server. This prevents the client from inadvertent interactions with any malicious applications that attempt to impersonate servers.

OO-DTE access control plug-ins rely on vendor-provided Secure Socket Layer (SSL) [DIERK] packages to authenticate clients and servers and protect traffic between them. SSL was selected as the

communications security mechanism for OO-DTE, primarily because it is the most widely supported security technology for CORBA.

The access control plug-ins also rely on SSL to convey to servers the authorization of clients, and to clients the authorization of servers. SSL uses X.509 certificates that are issued to each principal. Each X.509 certificate used with OO-DTE contains the principal's domain.[6] Consequently, a user authorized to act in multiple roles will have multiple certificates, each containing a different domain, and must designate the certificate appropriate to the role in which he or she is currently acting. This is analogous to a retail customer choosing from his or her wallet a credit card appropriate to (i.e., accepted by) the merchant with whom a purchase is sought.

SSL handshaking is performed whenever a client and server establish a connection. The handshaking process causes exchange and cryptographic verification of information contained in the certificates including the domains of the client and server. Each ORB retains this information for the lifetime of the connection and makes it accessible to the access control plug-in for use in authorization checking as described above.

Figure 2 shows the relationships among the OO-DTE components:

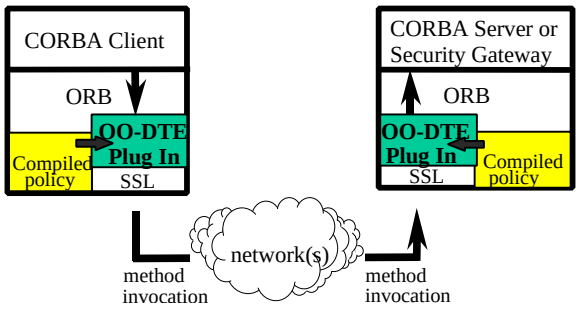

Figure 2 -OO-DTE Components

4.3. Per-Object Access Control

The OO-DTE plug-in makes access control decisions based on the OO-DTE type of the operation, the domain of the client (or server), and the definition of that domain as described in the DTEL++ policy. The type assignment for the operation is also described in the policy, as discussed in Section 3 above. The type depends on the operation, interface, and, when per-object access control policies are used, the object's name, e.g., /Books/Antique/1003. The operation is part of the request object passed to the plug-in. The interface name can be determined by invoking local methods on the target object, such as retrieving the Repository ID. Obtaining the object name, if present, required the development of additional, non-standard mechanisms.

Per-object access control relies on assigning unique identifiers (object names) to objects and organizing the object names in a hierarchical manner. CORBA deliberately avoids providing unique identifiers for objects, so implementing OO-DTE required inventing a specialized naming mechanism. There are several requirements for this mechanism:

- An object can have, at most, one OO-DTE name.
- There must be a means for the plug-in to determine the name of an object given a reference to the object, preferably without making a remote invocation.
- The names must be organized in a hierarchical manner to allow easy association with type templates.
- Object names must be assigned at object creation and thereafter be immutable. Objects cannot be named after creation.

An obvious candidate for naming objects is the CORBA Naming Service. However, the Naming Service does not provide a two-way association between objects and names. Given a name, it is possible to find the associated object using a Name Server. However, it is not possible to determine an object's name from a reference to the object. Furthermore, CORBA Naming provides for a many-to-one mapping between names and objects.

The OO-DTE prototypes use a combination of mechanisms to assign names to objects. First, the object name is bound to the object in a secure implementation of the Naming Service. The Naming Service provides structure to the naming hierarchy and ensures that object names are not re-used. Second, a 'stringified' form of the object name is stored in the *object key* portion of the object reference to provide the two-way association between the name and object. This allows the object name to be extracted from the object reference by the OO-DTE plug-in on the client side. Since object factories are responsible for creating and naming the object correctly, per-object access control requires a security-aware application. However, the impact is limited to the factory.

[6] In the initial prototype of Above Kernel OO-DTE, the Organizational Unit of the Distinguished Name contains the domain information.

4.4. Policy Distribution and Synchronization

OO-DTE takes a decentralized approach to authorization decisions in order to keep policy information close to OO-DTE enforcement mechanisms, which are resident in client and server plug-ins, and provide faster access decisions than could be obtained by consulting with a centralized access decision server. With this approach, however, all policy changes must potentially be propagated to multiple hosts to ensure consistent policy enforcement. The OO-DTE prototype provides a central point of control for policy changes and distribution. Policy update notifications are pushed from this point to individual OO-DTE hosts, providing loose synchronization of policy changes.

OO-DTE policy distribution uses CORBA methods for policy change notification and for transport of the policy updates. A *master policy server* accesses the authoritative copy of the policy. *Local policy servers* on each host register with the master policy server at startup. Applications (client and server programs) register with their host's local policy server at startup. When the policy administrator changes the master policy, an event notification propagates from the master policy server to each of the registered local policy servers, and then to the security plug-ins in the client and server ORBs. Local policy servers invoke methods on the master policy server to retrieve the updated policy and then copy it into local files, which are then read by the plug-ins of local client and server processes. In this way, policy changes are put into effect transparently to applications and 'on-the-fly', without stopping or restarting applications. For scaling to a large number of policy subscribers, policy distribution servers can be arranged into a hierarchy, with each server communicating policy changes only with its immediate superior and subordinates. All policy notifications and transfer operations are protected by SSL.

4.5. ORB Gateway and Multi-Protocol Object Gateway

OO-DTE components have also been integrated into two network security gateways: the *ORB Gateway* and its successor, the *Multi-Protocol Object Gateway*. Both Gateways are deployed on or behind firewalls where they intercept CORBA's Internet Inter-ORB Protocol (IIOP) traffic addressed to servers in the enclave. The Gateways perform control access in accordance with a DTEL++ policy before forwarding requests to the servers. If a Gateway denies a request, the request is not forwarded. Neither Gateway requires modifications to CORBA object references or IIOP messages. In the event that a remote client cannot present an X.509 certificate containing an explicit DTE domain, the Gateways can be configured to use other information about the client, including the client's IP addresses or the contents of other X.509 fields, to infer or synthesize a domain for use in access control decisions. Once a domain is established for that client, the rest of the access decision process proceeds as described above. The Multi-Protocol Object Gateway provides the same functions as the ORB Gateway but supports a subset of Java RMI in addition to CORBA IIOP and will be a focus of our future research.

5. Comparison to Access Control in CORBA Security

In this section we relate OO-DTE to the access control terminology and features described in CORBASec [CORB].

5.1. OO-DTE Types vs. Standard Rights Family

According to the CORBASec model, an *access policy* translates privilege attributes held by a principal (e.g., group membership) into *rights* and specifies which rights a principal must hold to invoke operations on CORBA objects; these are called *required rights*. Rights belong to *rights families*, which are simply sets of rights, like sets of access modes. CORBASec defines a *standard rights family* that consists of the rights "get", "set", and "manage". These correspond to read-only, update, and administrative access modes, respectively. CORBASec allows other rights families to be used, but discourages them.

While the standard rights family (get, set, manage) is simple and general, its usefulness seems quite limited. To illustrate, we will attempt to use it to protect the `Book` interface in the library application discussed in Section 3. Consider the requirement that patrons and librarians are allowed to reserve books, but only librarians are allowed to check out books. Both of the `Book` methods `reserve` and `checkOut` change the application's state. Hence, for both, the most appropriate required right in the standard rights family is "set". But if the "set" right is granted to both librarians and patrons, patrons will be allowed, inappropriately, to invoke the `checkOut` method. On the other hand, if only librarians are granted the "set" right, then patrons will be prevented, inappropriately,

from invoking the `reserve` method. Unfortunately, the standard rights family cannot readily address these simple and ordinary requirements.

The underlying problem is that the interface to an object often includes multiple "set" methods that update different parts of an object's state and should be accessible to different groups of principals. To adequately specify the required rights for an interface, a *distinct* right might be needed for each update method. Hence, the optimal number of update rights depends on the application and cannot usefully be reduced to the single "set" right provided by the standard rights family. By contrast, the OO-DTE "rights family" is simply the collection of types declared by the DTEL++ policy writer to suit the characteristics of the application, e.g., safe_t and restricted_t. The simplicity of the OO-DTE solution for the library application presented above illustrates OO-DTE's flexibility and effectiveness at solving this common problem.

5.2. Per-Object Access Control

OO-DTE provides named objects and type templates so that access control can be configured on a per-object basis when needed. CORBASec states, however, that "Required Rights are characteristics of interfaces, *not* instances. All instances of an interface, therefore, will always have the same Required Rights."[7] Consequently, it is not possible to directly specify per-object access control using CORBASec's Required Rights mechanism. To get around this restriction, the application architect is forced to put individual objects or collections of objects – all of which belong to the same interface – into separate *Security Policy Domains*. The specification defines a Security Policy Domain as "a set of objects to which a security policy applies for a set of security related activities and is administered by a security authority."[8]

Consider the use of per-object access controls to provide special handling of antique books in our library example. Addressing this handling requirement by using the CORBASec Required Rights necessitates placing antique books into a different Security Policy Domain than ordinary books. This seems awkward and artificial because both kinds of books are objects "to which a [single] security policy applies" and both are probably controlled and "administered by a [single] security authority". These quotes from CORBASec suggest that both kinds of books *should* belong to the same Policy Domain. Moreover, the administrative and programming ramifications of using multiple Security Policy Domains are unknown because CORBASec provides no guidance on 1) how Policy Domains are created, deleted, and configured, and 2) how objects are moved among domains. The concepts and implementation issues of Security Policy Domains are now being re-examined by the CORBASec community. An RFP seeking clarification of the issues surrounding Policy Domains is currently underway. For simple applications like our library application, forcing designers to use multiple Policy Domains seems cumbersome, risky, and unnecessary given that OO-DTE achieves the desired result without them.

5.3. Supporting and Exploiting Inheritance

DTEL++ uses a collection of wild-card rules to facilitate the assignment of types to methods and make these assignments as compact and intuitive as possible. In particular, DTEL++ exploits the inheritance hierarchy. When types are assigned to the methods of a base interface, by default, the inherited methods in all derived interfaces automatically inherit those type assignments. As a result, when derived interfaces are added to an application, few if any additional DTEL++ statements may be required, even if the base interface was complex and required numerous DTEL++ statements. In addition, DTEL++ provides a variety of optional flags to control and selectively override the propagation of type assignments through the inheritance hierarchy.

In the examples in CORBASec, Required Rights are shown as a table that enumerates the rights required for each method in *each interface*. Inheritance is not discussed. This suggests that when a derived interface is defined, it may be necessary to enumerate the required rights for each inherited method even if they are identical to the rights for the base interface's methods. This will be burdensome when deriving new interfaces from lengthy base interfaces. Furthermore, it is clearly unnecessary, as DTEL++ illustrates, and is conducive to errors and inconsistencies. The specification doesn't prohibit an implementation from exploiting inheritance to make the enumeration of rights more compact. But, it also fails to provide guidance or insights into the benefits, issues, or features that might be required.

6. Experience

The OO-DTE features described above have been implemented on three different ORBs (ILU, Orbix, and Visibroker) and three different operating systems

[7] See [CORB], Section 15.6.4, "Access Policies"
[8] See [CORB], Section 15.3.8, "Domains"

(DTE-enhanced BSD/OS, Solaris, and Windows NT) and in network security gateways. Although we have not tested OO-DTE with an operational CORBA application, we have used several non-trivial demonstration applications, including three that were written by other organizations; one of these was many tens of thousands of source lines in length.

6.1. Performance – Preliminary Results

We have recently begun collecting and analyzing OO-DTE performance data. Our initial target for testing is our plug-in interceptor for the Visibroker ORB, Version 3.2. This ORB and our plug-in for it are written in Java and are intended for use with CORBA programs written in Java. In our tests, we timed a repetitive set of operation invocations provided by a demonstration application written by another DARPA contractor. Start and stop times were collected on the client machine issuing the invocations. The start time was captured prior to invoking the first operation in the set, but after establishing a (secure) connection to the server. The stop time was captured after completing the set of operation invocations.

The operation that produced the most consistent test results is called "add_annotation". It sends 18 (application-layer) bytes of ASCII "annotation" data to a server and requests that the server replace the current annotation associated with a particular CORBA object. Timing data was collected for four configurations of this application: 1) no security or plug-in components; 2) SSL alone; 3) SSL with a "null" plug-in on the client and server; and 4) SSL with the OO-DTE plug-in on the client and server. This version of Visibroker supports only one configuration of SSL that provides confidentiality of messages (encryption) as well as authentication and integrity. For these tests, the client machine was a 133 MHz Pentium with 32 MB RAM. The server machine was a 166 MHz Pentium with 64 MB RAM. The results shown in Figure 3 below are based on computing the incremental timing differences between these four configurations, and dividing the differences by 1000, the number of times the operation was repeated between timings. This calculation provides the average time contribution per operation per component. Note that these numbers represent the time consumed by the *combination* of the client and server working together in a sequential manner.

Component	Average time per operation
• Application	4.09 ms
• SSL	2.95 ms
• Null plug-in	1.0 ms – 1.25 ms
• OO-DTE access check	0.02 – 0.10 ms
Total (average)	8.33 ms

Figure 3 - Time contributions for the "add_annotation" operation

As shown, OO-DTE access checks, without any optimization, are relatively fast, consuming approximately 1% of the total time for this lightweight operation. Using the upper bound of 100 microseconds for a pair of OO-DTE checks (one on the client and one on the server) each check requires 50 microseconds or less on slow Pentiums (166 MHz or 133 MHz). With performance tuning, it is likely that OO-DTE speed could be improved, though it's not clear that it would matter much, given how slow the Visibroker plug-in mechanism and SSL (in this mode) are by comparison.

We also ran a number of other tests involving other operations, larger data transfers (and hence more encryption) and varying amounts of simulated compute time on the server. No other experiment provided consistent data that was at odds with the above.

6.2. Observations

In our testing, most OO-DTE features worked as expected. Some of the features that appeared to be unimportant in the design phase proved quite useful after implementation. For example, automated policy distribution proved invaluable during "policy debugging" because it allowed us to distribute experimental policy changes quickly to multiple hosts, including our security gateways. Other features designed to support larger, more complex policies have had limited use so far, e.g., inheritance rules among the type templates used for per-object access control. Nevertheless, we still expect them to prove valuable for operational systems.

Overall, DTEL++ has proven to be simple, flexible, and easy to use as a policy specification language. The

DTEL++ policies we have experimented with have tended to be compact, usually no larger than their associated IDL files, and in some cases, much smaller. This is largely due to the automatic inheritance of assigned type bindings and the ease of establishing default type bindings for an entire CORBA interface or module.

Our experience using DCE[9] integrated with Orbix Security to secure CORBA applications has heightened our appreciation of OO-DTE features. Compared with OO-DTE, we perceive DCE as a set of low-level enforcement mechanisms that lack appropriate abstraction facilities, especially for OO. (It might be possible, however, to hide DCE mechanisms under a layer of better administrative tools.) In particular, DCE does not help one *organize* an access control configuration as a set of high level *patterns*, as is required to enforce role-based policies and other organizationally mandated policies. For example, when a derived class is created that should have the same access control characteristics as its base class, a DCE administrator must create a new ACL for the derived class that is a duplicate of the base class's ACL. DCE provides no general way to specify that both ACLs are instances of a common pattern. For distributed OO applications that use inheritance, this makes DCE seem conducive to error and difficult to maintain, particularly when access configuration changes need to affect a collection of classes uniformly. Another advantage of OO-DTE over DCE is that OO-DTE facilitates inspection, audit, and analysis of the access control configuration because a complete description of the current access control configuration resides in a single or small set of DTEL++ files. By contrast, auditing a DCE configuration requires opening and inspecting every ACL in the system.

A few aspects of OO-DTE did not work as well as hoped. The syntax for defining type templates and the inheritance relationships between type templates is somewhat confusing and will be refined. We also plan to experiment with a syntax that allows multiple interfaces per type template. Some DTEL++ semantics have proven ambiguous or difficult to implement in the compiler, for example, the semantics governing the propagation of default type assignments into interfaces with multiple inheritance. A rule is needed that specifies which set of defaults (if any) propagates into the derived class. One possibility is to have no default in cases that could be ambiguous, and instead require an explicit type assignment.

[9] Distributed Computing Environment from The Open Group

It is desirable to detect as many DTEL++ errors as possible at compile time because it is generally more difficult to detect and diagnose errors at runtime. Consequently, the DTEL++ compiler verifies that all module, interface, and operation names in a DTEL++ policy match identifiers in the associated CORBA IDL files and reports any mismatches. However, we have experienced other sources of DTEL++ errors that could potentially be detected prior to runtime by additionally cross-checking. First, names of DTE domains that represent user roles could be compared against the set of DTE domains known to the certificate authority that issues the X.509 certificates used with OO-DTE; the names of DTE domains in these certificates should be consistent with the DTEL++ policy. Second, a DTEL++ policy that uses per-object access controls assigns type templates to names in CORBA's object name space. It might be possible to detect naming errors in a DTEL++ policy prior to runtime by consulting with the CORBA naming service to validate the existence, availability, ownership, etc. of these names.

Another area for improvement is display of DTEL++ policies. DTEL++ was designed as an adjunct to CORBA IDL. Consequently DTEL++ is terse and contains little information already present in IDL. For example, DTEL++ does not explicitly describe the inheritance hierarchy nor does it enumerate all the operations that are part of an interface. While avoiding redundancy with IDL simplifies maintenance, it also means that neither IDL nor DTEL++ by itself presents a "complete picture" for a human administrator. A useful addition to OO-DTE would be a viewer tool that provides an integrated view of IDL and DTEL++. For example, it might enumerate all the operations in the interface, and for each, the DTE type assigned as a net result of defaults and explicit overrides in the DTEL++ policy. In a related development, researchers at Secure Computing Corporation have developed a graphical administrative tool that can generate compilable DTEL++ [THOM].

DTEL++ was originally conceived of as a collection of annotations that would be embedded in IDL. These annotations would be either stripped out by a DTEL++ preprocessor prior to compiling the IDL or disguised as IDL comments. The approach described here was adopted instead because it allows an application's IDL and DTEL++ policy to be changed independently in many cases, thereby minimizing the "ripple effect". For example, a policy can be fine-tuned by adding new roles without changing the files containing IDL. This is desirable because without special tools, a change in IDL files might trigger unnecessary recompilation of

IDL, recompilation of the generated stubs and skeletons, and relinking of the client and server programs that use them.

OO-DTE does not provide any features for delegation, i.e., features that allow a client to dynamically authorize an intermediary server to act on its behalf when invoking methods on a target server. The primary motivation for delegation is to limit the extent to which an intermediary server must be trusted. This is accomplished by allowing individual clients to grant subsets of their own privileges temporarily to the intermediary. While this is intuitively appealing, it is not clear to the authors that delegation significantly reduces the amount of trust that must be placed in an intermediary server in many circumstances. For example, when the intermediary concurrently supports multiple clients having diverse privileges, the intermediary must be trusted to use the correct set of privileges for each request it makes on a target server. Furthermore, the cryptographic overhead and administrative costs of delegation appear significant. Moreover, in order for a client to explicitly authorize each target service used by the intermediary, the client must know in advance how the intermediary is implemented, i.e., must know what services the intermediary will use to implement the services it offers to clients. This seems to violate the well-established software design principles of abstraction and information hiding. Because the benefits of delegation do not seem to outweigh these costs, the authors have no plans at this time to support delegation in OO-DTE.

6.3. Future Plans

OO-DTE research and development is continuing under DARPA funding. We plan to continue research in the following areas:

- Attribute Certificates - For compatibility with SSL, our current implementation appropriates the Organizational Unit field of an X.509 (identity) certificate and uses it to represent a principal's role. It would be preferable to pass this information separately in an attribute certificate.
- Role Authorization Database - We are currently developing an alternative approach for exchanging role authorization information as part of the IIOP message stream and validating it. The goal is to reduce the need to issue and revoke certificates when authorization changes occur, which may be frequently. In this approach, each application's plug-in validates its counterpart's requested role by comparing it against a local role authorization database. The database is distributed and updated via the same mechanisms that are used to distribute OO-DTE policy files.
- Policy Modules - Our current approach to policy distribution causes all DTEL++ policy subscribers to receive all policy updates. It would be preferable for each subscriber to receive only the policy updates it needs. Supporting this will require mechanisms for organizing a DTEL++ policy as a collection of separable modules that can be subscribed to independently.
- Java RMI - We plan to extend OO-DTE for use with Java RMI. One issue that has surfaced is that RMI supports overloaded method names. Because CORBA IDL does not, neither does DTEL++. DTEL++ will need to be extended accordingly for RMI.
- Role Instances - In some applications, there is a need to distinguish among different instances of the same role and extend different rights to each. For example, in a medical records application, there may be several instances of the primary physician role. Each instance should have access to the same fields in patient records, but for different patients. We plan to extend OO-DTE to support such requirements.

7. Summary

Although distributed object technology has matured greatly during recent years, the lack of practical, integrated security mechanisms, particularly for access control, remains an obstacle to its deployment in many application domains. It is often necessary to control access to individual objects and methods. In large systems, however, these can be so numerous that the resulting proliferation of access control information can be overwhelming. We have described Object Oriented Domain and Type Enforcement (OO-DTE), a research technology for organizing, specifying, and enforcing access control that has been prototyped and integrated with commercial ORBs and SSL. OO-DTE is an outgrowth of our earlier research into access controls for secure operating systems.

Our experience developing OO-DTE suggests that it has been successful in meeting its original goals:

- Object Oriented - OO-DTE is object oriented. DTEL++, its compilable policy language, resembles CORBA IDL, supports OO abstractions, and takes advantage of the inheritance hierarchy to

improve the conciseness and understandability of access control policies.

- Scalability and Manageability - DTEL++ provides wild-card techniques for assigning OO-DTE types to methods based on the inheritance, lexical name scoping, and object naming hierarchies. These techniques allow a few DTEL++ statements to control thousands of objects and methods. OO-DTE also provides mechanisms to automatically distribute policy changes to large numbers of clients and servers.
- Fine-grained Control - OO-DTE provides fine-grained control over individual objects via named type templates. Type templates allow distinct combinations of OO-DTE types to be assigned to individual objects or collections of objects whose names share subtrees of the object name space.
- Role-based Access Control - OO-DTE provides access control based on user roles. Roles, implemented as OO-DTE domains, can be defined in an application-specific manner via DTEL++. A user's authority to act in a role is currently represented by the value of a field in the user's X.509 certificate.
- Compatibility with Commercial Products - OO-DTE has been implemented as a plug-in module for Iona Orbix and Inprise Visibroker, the market leading ORBs.
- Transparency - The OO-DTE plug-in supports security-unaware applications by automatically invoking SSL authentication and access checks without direct participation by application code.

We have compared OO-DTE to the access control terminology and features of CORBASec and described what we believe are important advantages of OO-DTE over CORBASec's Required Rights and standard rights family. These advantages include the ability to 1) address a wider range of real-world access control requirements; 2) provide per-object access control within a single Security Policy Domain; and 3) express policies for derived classes in a more compact and understandable manner.

We have also identified several areas where OO-DTE could be improved or extended including adapting OO-DTE for use with Java RMI. Improvements and extensions of these kinds, combined with OO-DTE's designed-in scalability, should allow OO-DTE to mature over time from a research technology to a practical base of mechanisms suitable for commercial products and large-scale distributed object systems.

Bibliography

[BADG] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker. *"A Domain and Type Enforcement UNIX Prototype,"* USENIX Computing Systems Volume 9, Cambridge, MA, 1996.

[BALD] R. W. Baldwin, *"Naming and Grouping Privileges to Simplify Security Management in Large Databases"*, Proceedings of the IEEE Symposium on Security and Privacy, page 116 (May 1990).

[BOEB] W. E. Boebert and R. Y. Kain, *"A Practical Alternative to Hierarchical Integrity Policies,"* Proceedings of the 8th National Computer Security Conference, pp. 18-27, Gaithersburg, MD, September 1985.

[CORB] CORBA Services: Common Object Services Specification, Chapter 15, Security Service Specification, November 1997 Object Management Group, Inc.

[DENG] R. H. Deng, S. K. Bohnsle, W. Wang, A. A. Lazar, *"Integrating Security in CORBA-Based Object Architectures"*, 1995 IEEE Symposium on Security and Privacy, page 50.

[DOWN] D. Downs, J. Rub, K. Kung, C. Jordan, *"Issues in Discretionary Access Control"*, 1985 IEEE Proceedings of the Symposium on Security and Privacy, page 208.

[HU] W. Hu, DCE Security Programming, 1995, O'Reilly & Associates, Inc.

[DIERK] T. Dierks, C. Allen, The TLS Protocol Version 1.0, Internet Engineering Task Force, RFC 2246, January, 1999.

[KUHN] D. Kuhn, J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, *"Role-Based Access Control for the World Wide Web"*, 20th National Information Systems Security Conference, page 331.

[NICO] V. Nicomette and Y. Deswarte, *"An Authorization Scheme for Distributed Object Systems"*, 1997 IEEE Symposium on Security and Privacy, page 21.

[OSG] Orbix Security Guide, 1991, IONA Technologies PLC.

[RMISEC] Java RMI Security Extension, Early Look Draft, Sun Microsystems, Palo Alto CA, 1999 http://java.sun.com/products/jdk/rmi/rmi-security.pdf

[SALT] J. Saltzer and M. Schroeder, *"The Protection of Information in Computer Systems"*, IEEE Proceedings, 63(9), March 1975.

[SHER] D. Sherman, D. Sterne, L. Badger, S. Murphy, K. Walker, S. Haghighat, *"Controlling Network Communication with Domain and Type Enforcement"*, 18th National Information Systems Security Conference, October 1995, page 211.

[TALL] G. Tally, D. F. Sterne, C. D. McDonell, "Sigma Project: DTEL++ Language Specification", Trusted Information Systems, September 1998.

[THOM] D. Thomsen, R. C. O'Brien, J. Bogle. *"Role Based Access Control Framework for Network Enterprises,"* Proceedings of the 14th Annual Computer Security Applications Conference, pp. 50-58, December 1998 .

[WALK] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. *Confining Root Programs with Domain and Type Enforcement.* Proceedings of the 6th USENIX Security Symposium, San Jose, CA, 1996.