Security Engineering for Roles and Resources in a Distributed Environment

Computer Science & Engineering Dept.
191 Auditorium Road, Box U-155
The University of Connecticut
Storrs, CT 06269-3155
charlesp@engr.uconn.edu
Tel: 860.486.5582
Fax: 860.486.4817

Profs. Steven A. Demurjian and T.C. Ting
Computer Science & Engineering Dept.
191 Auditorium Road, Box U-155
The University of Connecticut
Storrs, CT 06269-3155
{steve, ting}@engr.uconn.edu
Tel: 860.486.4818
Fax: 860.486.4817

Abstract

One critical challenge to security for distributed applications (which consist of legacy, COTS, databases, clients, etc.) is managing access to available resources (and their APIs). An engineered solution is critical to insure that clients are restricted to select portions of resource APIs at different times and under specific conditions. This work offers a constraint-based role security model and enforcement framework for authentication, authorization, enforcement, and policy realization of distributed applications. Our approach utilizes middleware (e.g., CORBA and/or JINI), to federate users and their resources (e.g., APIs of Legacy, COTS, and databases) into a securely engineered solution that attains fine-grained security via permissions placed against resource APIs (methods) constrained by time and data values as defined for individual user roles. This allows the realization of a flexible security policy and provides the users with assurance. In this paper, we present a formal constraint-based role security model and corresponding enforcement framework (with prototype) that can serve as a strong basis for the security engineering of applications such as the Dynamic Coalition Problem.

1. Introduction.

Today's distributed applications typically consist of clients, legacy, COTS, databases, and other software artifacts interacting with one another using middleware (e.g., DCE [Rose92], CORBA [OMG95], DCOM [Micr95], or JINI [Arno99]) as a bridge. In this distributed environment, security realization is often an afterthought that depends on programmatic effort. For example, while CORBA has confidentiality, integrity, accountability, and availability services, there is no cohesive CORBA service that ties them together with authorization and authentication. Also, in JINI, the services of a resource are available to any and all clients without restriction. More importantly, with legacy and COTS systems, programmatic changes are not always possible (i.e., no source code). To integrate security, ad-hoc techniques are often employed that are prone to errors and inconsistencies. Consequently, we are left with poorly engineered solutions without information and/or security assurance. To address this critical concern, security must be a first class citizen in the design, prototyping, and integration processes for distributed environments.

The different constituent artifacts (software or hardware) within a distributed environment interact with one another via APIs, which are typically available (public) for any and all to utilize. In this situation, the emphasis is what a user playing a specific role can and cannot do with respect to the available APIs [Sandu99, Thom98]. This is true in our role-based approach to security for object-oriented models, which is focused on who can use which parts of artifact APIs based on role [Demu97], making it possible to enforce, at runtime, this custom behavior, in object-based [Demu97], agent [Demu00], and distributed [Demu01] settings. An end user playing a role via a client application can be restricted as to which methods can be invoked. Thus, the same client application will work differently at different times based on role. In this paper, our specific interest is in distributed applications that are composed of software/system resources (e.g., legacy, COTS,
databases, servers, etc.) which have services that are published (via APIs) for use by distributed application clients and components.

The intent of this paper is to significantly extend our prior effort [Demu01], which was limited to either granting or denying the method call of the resource based on role, to a more versatile and robust solution that permits the invocation of methods by roles based on allowable data values and valid time intervals. In the process, we introduce rigor into the security definition, management, and enforcement processes, providing an engineered solution.

Specifically, we formalize a constraint-based model for role security to support access to methods by role based on: signature constraints (SC), the invocation of a method based on allowable parameter and/or return values; and, time constraints (TC), the invocation of a method based on time. The constraint-based model exists within a comprehensive enforcement framework, which can dynamically insure that access by clients to resources satisfies all defined security privileges and constraints. Our objective is to engineer an architectural model, which can be used in a distributed environment. Specifically, as given in the bottom-half of Figure 1, a distributed application is comprised of various software systems and artifacts interacting with one another using middleware (and lookup services). To this environment, we propose to add a robust collection of security capabilities (top-half of Figure 1) which seamlessly provide customized role-based access to artifact APIs.

The remainder of this paper is organized as follows. In Section 2, we formally define a constraint-based distributed security model with a role-based focus. In Section 3, we detail the enforcement framework and prototyping efforts. In Section 4, we provide a discussion of the dynamic coalition problem [DARP00], which requires dynamically changeable security in an efficient, scalable, and reliable fashion. In Section 5, we conclude and discuss ongoing work.

2. Distributed Security Model.

In this section, we formally define our constraint-based distributed role-security model. In Section 2.1, we discuss a middleware lookup service model. In Section 2.2, we define the concepts of resources, services, and methods. Finally, in Section 2.3, we define security constraints, time constraints, user roles, and authorizations.

2.1. Using Lookup Middleware.

A Lookup Service, as supported in middleware such as CORBA, JINI, DCOM, etc., is a clearinghouse for resources to register services and for clients to find services. A lookup service allows stakeholders to construct distributed applications by federating groups of users (clients) and the resources that they require [Arno99, Demu01]. With any lookup service, it is key that the services are registered or they will not be readily available. Resources register services provided for use by a person, program (client), or another resource, including a computation, a persistent store, a communication channel, a software filter, a printer, and so on.

Figure 2 illustrates the interactions of a lookup service, client, and resource in a distributed resource environment (DRE). In Figure 2, the CourseDB resource joins the lookup service, by registering its services (methods). Each service consists of methods (e.g. AddCourse) that are provided for use by clients (and other resources). One limitation of this process is that once registered, all of the resource’s services are available to all clients, i.e., there is no security. Figure 2 also illustrates the steps that are taken when a client requests a service (AddCourse) from a resource (CourseDB).
2.2. Resources, Services, and Methods

This section introduces definitions for the resources that comprise a distributed application. Note that Definitions 1, 2, and 3 have been reformulated from [Demu01]; Definition 4 is new:

**Definition 1:** A distributed application consists of \( m \) software/system resources (e.g., a legacy, COTS, database, Web server, etc.) and is defined as the set: \( \{ R_i \mid i = 1K m \} \). A unique resource identifier (R-Id) differentiates between replicated resources.

**Definition 2:** Each resource \( R_i, i = 1K m \), is composed of \( n_i \) services, defined as the set: \( \{ S_{ij} \mid j = 1K n_i \} \). A unique service identifier (S-Id) distinguishes the services of a resource.

**Definition 3:** Each service \( S_{ij}, j = 1K n_i \), of resource \( R_i, i = 1K m \), is composed of \( q_{ij} \) methods, defined as the set: \( \{ M_{ijk} \mid k = 1K q_{ij} \} \) where each method is similar to an object-oriented method, and represents a subset of the functionality provided by the service. A unique method identifier (M-Id) distinguishes the different methods of a service. Note that the triple (R-Id, S-Id, M-Id) is unique.

**Definition 4:** The signature of a method \( M_{ijk} \), for some \( i = 1K m, j = 1K n_i, k = 1K q_{ij} \) of service \( S_{ij} \) of resource \( R_i \) is unique, and consists of following:

- a parameter list, \( P_{ijk} \), which contains a list of pairs of parameter name and parameter type (which may be null) and assumes pass-by-value parameter passing, and
- a return type, \( RT_{ijk} \), which may be null, to represent the single value (or object) returned by the method.

Given the pass-by-value assumption, a method can return at most one value via its return type.

To serve as a basis for discussion, we utilize a university application where students can query course information and enroll in classes, and faculty can query and modify the class schedule. In Figure 3, we present a University Database Resource and its two services, Read and Modification, that are respectively used to query and change the database. The \( Token \) parameter provides the means to uniquely identify every client with an active session. The Read service is used by faculty and students and contains various get methods to query the database of courses. The Modification service allows students to modify their schedule (registerCourse and dropCourse methods) and faculty to modify the course offerings (addCourse, removeCourse, and updateEnroll methods). The updateEnroll method is called by registerCourse and dropCourse when a student adds or drops a course.

**Figure 3: Services and Methods for the University DB Resource.**

```
Read Service with Methods:
String getAllClasses (Token);
String getRegisteredCourses (Token, StudentName);
Vector getClasses (long Token, Semester);
Vector getClassDescription (Token, Course);
Vector getPreReqCourses (Token, Course);
Vector getVacantClasses (Token, Semester);

Modification Service with Methods:
boolean addCourse (Token, Course);
boolean removeCourse (Token, Course);
boolean updateEnroll (Token, CourseNumber, UpdateChoice, NewValue);
boolean registerCourse (Token, Course, StudentName);
boolean dropCourse (Token, Course, StudentName);
```

2.3. Roles, Constraints, and Authorizations

In this section, we provide the critical definitions for our constraint-based role-security model, which we use to support the realization of our security framework for a distributed environment (see Section 3). Note that, Definitions 6, 7, 8, 9, and 10 have been added to [Demu01].
Definition 5: A user role, UR, is a named entity with a unique UR identifier (UR-Id) that represents a specific set of responsibilities against an application. These responsibilities will be realized as the specific resources, services, and methods that have been granted to a user role based on the assessment of the role's scope and capabilities by a security officer.

As an example, user roles for faculty, department head, undergraduate student, graduate student, etc., are all representative of the different types of end users. The granularity of roles may also be more fine-grained at the user level, such as, transcript-issuer, or task oriented, such as advise-student, record-grades, etc. From a privilege perspective, a user role will be granted access to resources, services, and methods. For example, a CSEFaculty role may be granted access to the Read service of the University DB Resource (Figure 3), or to methods such as addCourse or removeCourse from the Modification Service.

We now define and discuss signature and time constraints in Definitions 6 and 7.

Definition 6: A signature constraint, SC, is a boolean expression that is defined on the signature of some method $M_{ijk}$ named $N_{ijk}$ for some $i = 1 K$, $m = 1 K$, $n = 1 K$, $k = 1 K$, $q_{ijk}$, of a service $S_{ij}$ of a resource $R_i$, to limit the allowable values on the parameters, $P_{ijk}$, and the return type, $RT_{ijk}$. The boolean expression has the form: (return-type constraint) and (parameters constraint) where either or both constraints can be null. The return-type constraint limits the value of what is returned as a result of the method call, thereby constraining $RT_{ijk}$. The parameters constraint is a boolean expression of relational expressions (e.g., Course $>$ CSE203) combined using the operators: AND, OR, and NOT.

SCs limit the conditions under which a method of a service for a resource may be invoked. For example, a CSEFaculty UR can use addCourse and removeCourse methods of the Modification service, but would need a SC (Course $\geq$ CS101 and Course $\leq$ CS499) for CS courses only. Note that SCs are independent from database integrity constraints (responsibility of the resource) and programming types (responsibility of the compiler/runtime environment). Thus, methods are no longer in an off/on state, but are off/on based on a specialization of the parameter/return values, similar to specialization of constraints [Zdon90].

Limiting the execution of methods based on a time period, controls when a user (playing a role) can execute a particular method, leading to:

Definition 7: A time constraint, TC, is an expression that is defined to represent a discrete period of time (i.e., days or time period in GMT) under which a method can be invoked. The TC expression can involve relational expressions (e.g., Date < March 19, 2001) that can be combined using the operators: AND, OR. The TC expression can also have the values: always (unconstrained by time - the default value) and never (prohibited at any time). We define:

$$TC = \{e | e = \text{“never”} \text{ or } e = \text{“always”} \text{ or } e = \text{boolean expression}\}.$$ 

Our notion of TC is similar in concept to [Ahn00, Bert96], but differs since their work focuses on controlling access to objects based on time, while our effort controls access to a resource, service, and/or method based on time. In a university application, during certain timeframes, only students with certain roles are allowed to register, and would do so via a client application that calls the registerCourse method (see Figure 3). During weeks one and two, only students with a CSEUndergrad UR can register for courses, resulting in the TC: (Date $\geq$ April 1, 2001 and Date $\leq$ April 14, 2001). During weeks three and four, students with a CSEGrad UR can register, resulting in the TC: (Date $\geq$ April 15, 2001 and Date $\leq$ April 30, 2001). Thus, the registerCourse method must always be available for use, although at certain times, its use will be limited. As a complementary example, prior to the period, a CSEDeptHead can add courses to the schedule (addCourse method) with the TC: (Date $\geq$ January 1, 2001 and Date $\leq$ March 31, 2001). SCs and TCs can be combined to yield concrete situations for access by an UR to a method. For instance, for a CSEDeptHead UR, we limit the courses that can be added to the schedule and the time period when courses can be added, leading to SC and TC on the addCourse method:
SC: (Course ≥ CSE101 and Course ≤ CSE499)
TC: (Date ≥ January 1, 2001 and Date ≤ March 31, 2001)

In practice, all students are constrained based on their academic standing and major field of study.

Using Definitions 1 through 7, we can define which privileges (i.e., resources, services, and methods - see Definitions 1, 2, and 3) can be authorized to user roles under certain SCs/TCs:

**Definition 8:** Assume a distributed application consists of resources, services, and methods:

\[
\{ R_i, S_j, M_{ijk} \mid i = 1K m, j = 1K n, k = 1K q \}.
\]

Then, a security privilege tuple can be defined which contains a specific resource, service, and/or method (with optional signature constraint) that has been authorized (granted) to a user role UR according to an (optional) time constraint: \([UR, TC, R_i, S_j, [M_{ijk}, SC_{gk}]]\).

**Definition 9:** Assume a distributed application consists of resources, services, and methods:

\[
\{ R_i, S_j, M_{ijk} \mid i = 1K m, j = 1K n, k = 1K q \}.
\]

Then, a security privilege tuple set, \( \rho \), contains all of the resources, services, and methods that have been authorized (granted) to a UR:

\[
\rho = \{ [UR, TC, R_i, S_j, [M_{ijk}, SC_{gk}]]\},
\]

for all triples of \( i, j, \) and \( k \) granted to UR

Under this notation, a security officer can grant a UR access to all of a resource’s services and their methods via the privilege tuple \([UR, TC, R_i, *, [*]]\). Alternatively, a UR can be granted access to all methods of a service, by the privilege tuple \([UR, TC, R_i, S_j, [*]]\). In all situations, TC can be equal to an actual constraint, or be set to “always” or “never” (see Definition 7). In addition, if a SC is set to true, then the allowable values of the signature are not limited. To illustrate the concepts, in Figure 4, there are CSEDeptHead, CSEFaculty and CSEUndergrad URs, and their respective privilege tuple sets defined against the University DB resource.

The next step is to authorize a UR to an end user. This leads us to define the following:

**Definition 10:** A user, \( U \), is identified by a unique user identifier (User-Id) and is authorized to play one or more URs in an application. A user must always play exactly one role at any point during an active session, but is able to change roles during a session.

**Definition 11:** A client, \( C \), represents an authorized user \( U \) utilizing a client application, and is uniquely identified during a specific session via a system generated Token composed of a four-tuple: \([User-Id, UR-Id, IP-Address, Token-Creation-Time]\). For a given session, the IP-Address is always unique; a client computer connecting to a network has the same IP address until the connection is severed, which would signify the end of a session.

Figure 4 also contains users (Harris, Smith, Jones) authorized to play different roles, and a sample client Token.

**3. An Enforcement Framework and Prototype for Distributed Security**

In this section, we present an enforcement framework and prototype for distributed role security, which enforces selective access of clients to resources, as shown in Figure 1. In the top half of Figure 1, we represent the security-related clients and resources that comprise our enforcement framework, namely: the Unified Security Resource (USR) to be discussed in Section 3.1; the security enforcement actions that occur for an active client to be examined in
Section 3.2; the process from the perspective of the resource registering its services for secure access to be considered in Section 3.3; and, finally, the prototyping of the framework and model, along with security management and administrative tools, to be reviewed in Section 3.4.

3.1. The Unified Security Resource (USR)

The Unified Security Resource (USR) consists of three sets of services: Security Policy Services managing roles and their privileges; Security Authorization Services to authorize roles to users; and, Security Registration Services to identify clients and track security behavior. The USR is a repository for all static and dynamic security information on roles, clients, resources, authorizations, etc., and is organized into a set of services, as given in Figure 5.

![Figure 5: The Services of USR.](image)

Security Policy Services are utilized to define, track, and modify user roles, to allow resources to register their services and methods (and signatures), and to grant/revoke access by user roles to resources, services, and/or methods with optional time and signature constraints. These services are used by a security officer to define a policy, and by the resources (e.g., database, Java server, etc.) to dynamically determine if a client has permission to execute a particular [resource, service, method] under a time and/or signature constraint. There are five different Security Policy Services: Register for allowing a resource to register its services for secure access to its services; Query Privilege for verification of privileges; User Role to allow the security officer to define and delete user roles; Constraint to allow time and signature constraints to be defined by the security officer, and for these constraints to be dynamically verified at runtime; and, Grant-Revoke for establishing privileges in accordance to Definition 9.

Security Authorization Services are utilized to maintain profiles on the clients (e.g., users, tools, software agents, etc.) that are authorized and actively utilizing non-security services, allowing a security officer to authorize users to roles. There are two services: Authorize Role for the security officer to grant and revoke a role to a user with the provision that a user may be granted multiple roles, but must play only a single role when utilizing a client application; and, Client Profile for the security officer to monitor and manage the clients that have active sessions. Security Registration Services are utilized at start-up for identity registration (client id, IP address, and user role), which allows a unique Token (see Definition 11) to be generated for each session of a client. Finally, the Global Clock Resource (GCR) is used by Security Policy Services to verify a TC when a client (via a UR) is attempting to invoke a method, and Security Registration Services to obtain the time, which is then used in the generation of a unique Token.

3.2. Security Enforcement: Client Application

The processing required by a client application is best illustrated with an example, which is given in Figure 6. In the first step in the process, the user of the University Client must be authenticated to play a particular role. To do so, the University Client registers with USR via the Register_Client method (step 1), which must verify the user role (steps 2 and 3), and return a generated Token via the Create_Token method (step 4). To generate the Token in step 4, the Security Registration Service interacts with GCR (steps not shown) in order to construct a Token which consists of User-Id, UR-ID, IP address, and creation time. This token is unique even if the user has multiple active sessions of same role on one machine.

Assuming that the registration and Token generation was successful, the user can then attempt to utilize services from the UnivDB Resource. The University Client consults the lookup service for the desired [resource, service, method] (step 5) which returns a proxy to RegisterCourse, allowing the method to be
invoked (step 6) with the parameters Token, CSE230, and Harris. The UnivDB Resource has two critical steps to perform before executing RegisterCourse. First, the UnivDB Resource verifies that the Client has registered with the security services (steps 7 and 8). If this fails, a negative result is sent back via the RegisterCourse result (step 11). If this is successful, then the UnivDB resource must perform a privilege check (privileges assigned by role) to verify if the user role can access the method limited by signature constraints and/or time constraints (both may be null). This is done in step 9, with two other method calls from Figure 5:

9a CheckTC (Token, UnivDB, Modification, RegisterCourse);
9b CheckSC (Token, UnivDB, Modification, RegisterCourse, ParamValueList).

The CheckTC method interacts with GCR to verify that the current time is within the limits of the time constraint (if present). If the privilege check is successful (step 10) then the method executes as called and the result (registering Harris for CSE230) is returned as a success in step 11. Otherwise, the result (step 10) denies the registration via step 11. Note that the signature constraint is verified in two phases. The parameters constraint (if present) must be checked prior to method invocation, while the return-type constraint (if present) must be checked after execution and before the result has been returned.

3.4. Security Prototype (JINI and CORBA).

This section reviews the prototyping efforts for our constraint-based model and enforcement framework. As shown in Figure 7, we have designed and implemented the entire security framework as given in Figure 1, which includes the USR, administrative and management tools, and global clock resource which are all capable of interacting with either CORBA or JINI as the middleware. The current version of USR implements all of the services from Figure 5. To verify the security framework, we have also prototyped two distributed applications, consisting of a Java client and associated database resource. We have utilized a university application (a University DB resource and client consistent with the examples in the earlier parts of this paper) and have also prototyped a hospital application where a client can access information in a Patient DB (Figure 7). The hospital portion of Figure 7 (left side) interacts with CORBA as the middleware; the university portion (right side) uses JINI. Note that the middleware lookup service is transparent to the user; however, a client could be constructed to use CORBA and/or JINI depending on which one is active in the network.

From a technology perspective, the university application in Figure 7 (right side) is realized using Java 1.3, JINI 1.1, Windows NT 4.0 and Linux, and Oracle 8.1.7. The hospital application (left side) uses the same technologies except for Visibroker 4.5 for Java as middleware. Both of the resources (Patient DB and University DB) operate
in a single environment using the same security system transparently, and both are designed to allow them to register their services with both CORBA and JINI. The University DB Java Client allows students to query course information and enroll in classes, and faculty to query and modify the class schedule. The Patient DB Java Client supports similar capabilities in a medical domain.

The security officer can also inspect and monitor security via the tracking capabilities of SPC. A typical security tracking history is shown in Figure 10. The results track status of the attempted access (Success/Error) of the method (method-id) of a resource (resource-id) by the particular user (user-id) playing a specific role (role-id).

The security officer can also inspect and monitor security via the tracking capabilities of SPC. A typical security tracking history is shown in Figure 10. The results track status of the attempted access (Success/Error) of the method (method-id) of a resource (resource-id) by the particular user (user-id) playing a specific role (role-id).

In addition, in Figure 7, there are administrative and management tools, namely: the Security Policy Client (SPC) manages user roles by granting/revoking privileges (TCs, resources, services, methods, and SCs); the Security Authorization Client (SAC) authorizes roles to end users; a Static Analysis Tool (SAT) (not shown) analyzes Java source code of resources to track nested method invocations; and, a Global Clock Resource (GCR) supports time-constrained access to resources (and their services/methods) by role. In the remainder of this section, we focus the discussion on the three tools (SPC, SAC, and SAT) to illustrate the techniques and processes that are utilized to engineer the security for a distributed application using our model and enforcement framework.

The Security Policy Client (SPC) in Figure 8 can be used to define and remove user roles, and to grant and revoke privileges (i.e., time constraints, resources, services, methods, and/or signature constraints). The SPC is a set of tabbed panels for each major function of the GUI, as shown in Figures 8 and 9. Figure 8 illustrates the assignment of services to resources and Figure 9 illustrates the use of SPC to define time constraints and signature constraints.

The Security Authorization Client (SAC) in Figure 11 supports authorization of role(s) to users. A user may hold more than one role, but can only act in one role at a time. Playing multiple roles at the same time may lead to conflicts if different SCs and/or TCs are defined on the same resource, service, or method. If more broad access is desired, a role with expanded capabilities can be defined to encompass privileges of multiple roles without SC/TC conflicts. The SAC is also a set of tabbed panels, shown in Figure 11, and demonstrates the ability to create a new User, which has an identifier, password, a time constraint (begin/end date) that indicates the valid time interval for the user, and a text description.
Finally, the Static Analysis Tool (SAT), is utilized by the security officer to analyze source code of Java resources, allowing us to track not only the method on a resource that has been directly authorized to a role, but also the other resources (and services/methods) that are called. Given the directory location of a Java source code, SAT analyzes a class by inspecting all of the method definitions to find any other method called inside the one under inspection. The result, in Figure 12, tracks the information on the method being analyzed, the methods that are invoked by the method, and the user roles assigned to the method.


In this section, we introduce the Dynamic Coalition (DCP) in detail, and discuss the utility, and feasibility of our model (Sections 2 and 3) in support of the DCP and other advanced applications. Additionally, we seek to define the security engineering of this distributed application, where security is necessary but often illusive. To meet our objectives, in Section 4.1 we define the DCP, and in Section 4.2 we introduce a real-world instance of the DCP, the Global Command and Control System (GCCS). In the process, we have employed SSE-CMM [SSE-CMM] to delineate and determine the security objectives of GCCS.

4.1. Dynamic Coalition Problem.

The dynamic coalition problem starts with a crisis. A crisis is any situation requiring national or international attention as determined by the President of the United States (in our case) or the United Nations. A crisis can be a natural disaster (earthquake), humanitarian relief (refugee camps), international incidences (spy plane), war (Gulf War), or combat operations other than war (Bosnia). A coalition is simply an alliance of organizations: military, civilian, international or any combination. A dynamic coalition can be defined as a coalition that is formed in a crisis with the only concern being the most effective way to solve the crisis. Generally, coalitions are formed quickly with little or possibly no regard to the composition of or the political interrelationships of the participants. The Dynamic Coalition Problem (DCP) can be defined as the inherent security, resource, and or information sharing risks that
occur as a result of the coalition being formed quickly, yet still finding information and resource sharing a necessity for crisis resolution [Spri00]. The resultant security policy must minimize access to only that essential information needed to complete the task in an easy, efficient, scalable, and reliable way. In the event of a crisis that requires combat operations, where classified information is used, the DCP takes on another dimension, by requiring mandatory access control.

4.2. Global Command and Control System.

The Dynamic Coalition Problem is a complex and difficult problem. Since the United States and its military are often called upon in a crisis to supply necessary goods and services, or a unique capability quickly, there must be a system to coordinate this action. The U.S. Military uses the Global Command and Control System (GCCS) to manage such activities in a joint and combined environment. Joint refers to the use of more than one branch of the Armed Forces (Army, Navy, Air Force, Marines, or Coast Guard) and combined means the participation of more than one country. GCCS is the command and control system of choice as determined by the Chairman, Joint Chiefs of Staff [GCCS]. Unfortunately, the GCCS does not provide for all of the requirements and conditions of a Dynamic Coalition.

GCCS provides a local commander with operational awareness of the situation in near real-time through an integrated set of resources. The GCCS system provides information-processing support to planning, mobility, sustainment, and messaging by bringing together 20 separate automated systems with several additions planned [GCCS]. There are several problems with the current systems, which we look to address with our security model. Note that we used the SSE-CMM, specifically the Security Engineering Processing Areas, to examine a real-world environment and determine the security policy needs. Below, we outline the security objectives that are required for a dynamic coalition:

- Ability to federate groups of users quickly and dynamically in response to a crisis.
- Bringing together resources (e.g., COTS, databases, legacy systems, etc.) without modification for usage in support of the crisis.
- Dynamically realizing and managing a security policy during simultaneous crises.
- Identifying users by their roles to finely tune their access in support of crisis.
- Authorize, authenticate, and enforce a scalable security policy that can be managed and changed in response to coalition needs.
- Provide a distributed security solution in support of DCP that is portable, extensible, and redundant for survivability.
- Offering robust security policy definition, management, and introspection capabilities that are able to track and monitor system behavior and activities of users.

There are a number of distinct characteristics and capabilities of our model/framework that strongly advocate its relevance for DCP, in general, and GCCS, in particular. First, multiple user roles are not currently considered. Individual users have a unique profile that allows access to resources within GCCS. This profile is determined by their role and supervisor input. Control of profiles is effective in a static environment, but in a crisis, where roles are changed or added quickly, this is very inefficient. It is too time intensive for the host administrator to rebuild new profiles. If a resource is added to GCCS, user profiles will need to be re-built. Our security model incorporates user roles as defined in Section 2.3, Definition 5. Using roles eliminates the profile change requirement by allowing the security officer to change the characteristics of a role or add roles to users dynamically. Changing a role would impact every user assigned that role, which is more efficient than changing user profiles. In addition, the host administrator will not need to be involved.

Second, time constraints (Definition 7) for users or roles are not considered in GCCS. When an individual is assigned to the organization, the profile is provided for an indefinite period of time. However, most often, a soldier is only assigned to a crisis for 90 days or less. In order to limit the use of GCCS for a fixed duration, time constraints can be utilized. Time constraints are also relevant for GCCS resources. Often times, a resource's usage is restricted to fixed time windows to facilitate database updates or resource allocation. This is the case with GCCS’s Joint Operations Planning and Execution System. Junior Planners schedule movement of equipment by air at least 14
days in advance. If an airlift is required inside of a 14-day window, only Senior Planners can make adjustments, and that is a different role, with different constraints.

A third shortcoming with GCCS is the inability to impose constraints on resources. Our model uses signature constraints (Definition 6) to achieve maximum granularity on the way that resources can be used. For example, the common operational picture (COP) is a capability of the GCCS. COP provides a near real-time mapping of all deployed units worldwide. Signature constraints using map coordinates can limit the map view to just the crisis area for a specific user role. This, for example, would limit a non-U.S. coalition partner to viewing force positions only in the area of concern. This allows for the user to do their job, but provides no other additional, and potentially sensitive, information.

Finally, while it is difficult to manage the GCCS in a joint environment with just U.S. forces, it is even more complicated to use GCCS in a multinational, distributed environment. Currently, in multinational crisis situations, there is no dynamic way to effectively bring users and their automation assets (resources) together in an efficient way. Security systems need to allow for quick administration, but still constrain non-U.S. users from committing security policy violations. By using middleware services like JINI or CORBA, resources from coalition partners can be federated with the GCCS to make it a more robust and flexible coalition system. Our proposed enforcement framework (Section 3) would allow for the management of federated resources and constrain users to security policy limits, limits that also need to be flexible. Our interest in GCCS is to investigate techniques to secure this system in a manner that would make it a coalition asset and respect both coalition and U.S. security policies.

5. Concluding Remarks and Future Work

In this paper, we have examined the realization of role security for a distributed resource environment via a constraint-based model and an accompanying enforcement framework. We formally defined a model that included resources, services, methods, and signatures (Section 2.2), and user roles, signature and time constraints, and security privilege tuple sets (Section 2.3). This model can be realized in an enforcement framework, consisting of a unified security resource (Section 3.1) that is utilized by clients for registration and dynamic enforcement of security requirements (Section 3.2) and by resources to register their services (and methods) for secure access (Section 3.3). Our approach has been incorporated into a working prototype (Section 3.4) using JINI and CORBA as middleware, which also has administrative and management tools (SPC, SAC, and SAT). The use of SSE-CMM was key in defining and examining the Dynamic Coalition Problem (Section 4).

The ongoing prototyping and planned research investigations to advance the work presented in this paper are quite extensive.

- One topic is integrating mandatory access control (MAC) features as specified by the Bell LaPadula Model and required by DoD directives. We are using existing time and signature constraints and the concepts of [Marv99], where patient information is protected using sensitivity levels to realize MAC features. Work by [Josh01] on the use of MAC in an object-oriented database, and [Irvi01] on data integrity in a multi-level secure environment, are also relevant.
- There is an effort on integrating this model into an agent environment to allow agents to securely interact with resources. An agent is a user, but authentication, authorization and enforcement may need to be different.
- Another topic area is establishing user constraints to develop a role deconfliction capability. Constraints for consistency [Bert99] and mutual exclusion [Essm98] will provide valuable insight to this topic.
- A pair of graduate students is examining delegation of authority (users delegating roles to other users) with relevant work found in [Berr99]. Delegation is an important security policy issue since a user is given authority to delegate roles without intervention.
- Finally, long term, we will consider the role of cryptography in our approach [Bash01, Dalt00, Fox96, Oppl97]. Proper use of cryptography needs to be part of any final implementation.
Acknowledgements: The authors acknowledge the efforts of the M.S. students who have worked on this project: F. Gao, Q. Jin, M. Lohawala, J. Ma, J. Nam, N. Paranjape, Z. Qian, and D. Wang.

References


