Toward Realizable Restricted Delegation in Computational Grids[1]

Geoff Stoker, Brian S. White, Ellen Stackpole, T.J. Highley, and Marty Humphrey

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
{gms2w, bsw9d, els2a, tlh2b, humphrey}@cs.virginia.edu

Abstract. *In a Computational Grid, or Grid, a user often requires a service to perform an action on his behalf. Currently, the user has few options but to grant the service the ability to wholly impersonate him, which opens the user to seemingly unbounded potential for security breaches if the service is malicious or errorful. To address this problem, eight approaches are explored for realizable, practical, and systematic restricted delegation, in which only a small subset of the user's rights are given to an invoked service. Challenges include determining the rights to delegate and easily implementing such delegation. Approaches are discussed in the context of Legion, an object-based infrastructure for Grids. Each approach is suited for different situations and objectives. These approaches are of practical importance to Grids because they significantly limit the degree to which users are subject to compromise.*

1 Introduction

A Computational Grid, or Grid is a wide-area distributed and parallel computing environment consisting of heterogeneous platforms spanning multiple administrative domains. Typical resources include supercomputers, PCs, archival storage, and specialized equipment such as electron microscopes. The goal of a Grid is to enable easy, fast, and inexpensive access to resources across an organization, state, country, or even the world. In the general model, individual users will be able to contribute to collective Grid services by creating and publicizing their particular service, for example, the specialized scheduler AppLeS [1]. Users could request action from this scheduler (perhaps at a cost) if they believed the output of the scheduler would be more efficient or predictable than some default scheduling mechanism.

In a Grid, a user may need to ask a service-providing object[2] to perform an operation on her behalf. For example, a scheduler may need to query an information service to determine on which machines a user is allowed to and has enough

[1] This work was supported in part by the National Science Foundation grant EIA-9974968, DoD/Logicon contract 979103 (DAHC94-96-C-0008), and by the NASA Information Power Grid program.

[2] For ease of presentation, an *object* is chosen as an arbitrary abstraction of an entity in a Grid.

allocation to execute a job. The scheduler does not have the privilege to ask the information service directly, because the information service protects its potentially proprietary information from unauthorized access. The information service would provide a user's information to a scheduler if it was acting on that user's behalf.

The conventional approach when a user must ask a service to perform some operation on her behalf is to grant *unlimited delegation*, which is to unconditionally grant the service the ability to impersonate the user. To date, restricted delegation is not used in emerging Grids because it is too difficult to design, implement, and validate except in very limited, *ad hoc* cases. While unlimited delegation is a reasonable approach in which all services can be wholly trusted by the users who wish to invoke them, it is clearly not scalable into general-purpose Grids. For delegations within a Grid, *the crucial issue is the determination of those rights that should be granted by the user to the service and the circumstances under which those rights are valid*. Delegating too many rights could lead to abuse, while delegating too few rights could prevent task completion.

Under *unlimited delegation* it is possible for a rogue object, to which unlimited rights have been delegated, to trick a trustworthy object into executing a service that it would otherwise not perform. The problem addressed in this paper is how to restrict, or eliminate, the potential security breaches possible when one object misplaces trust in another object. The problem does *not* focus on what an untrustworthy object can do directly, but rather what the untrustworthy object can get *other* trustworthy objects to do. There is a separate issue regarding how an initiator validates the correctness of a target's response, which is not directly addressed by this paper. While we acknowledge that a malicious object can "lie" in response to a request, we strive to ensure that that malicious object cannot obtain and misuse another object's rights.

There are two main contributions of this paper. First, we present a number of approaches for restricted delegation that more fully support realizable policy than existing approaches. "Realizable" in this context refers to the ability of one user to *define* and *implement* a policy by which to grant a limited set of privileges to another principal. By creating and supporting policies for restricted delegation that can be more easily determined, defined, and enforced, a number of security vulnerabilities can be reduced and even eliminated in complex computing environments such as Grids. Second, we explore the feasibility and implementation of these policies within Legion [8], which is mature software that provides a secure, fault-tolerant, high-performance infrastructure for Grids.

Section 2 gives a brief overview of Legion and its security mechanisms. In Section 3, we discuss general approaches for determining the rights to delegate when one object invokes the services of another and the applicability of each approach to Legion. Section 4 contains related work and Section 5 concludes.

2 Legion Security Overview

This work is presented in the context of Legion, an object-based Grid operating system that harnesses heterogeneous, distributed, and independently administered resources (e.g. compute nodes), presenting the user with a single, coherent

environment [8]. Unlike traditional operating systems, Legion's distributed, extensible nature and user-level implementation prevent it from relying on a trusted code base or kernel. Further, there is no concept of a *superuser* in Legion. Individual objects are responsible for legislating and enforcing their own security policies; Legion does provide a default security infrastructure [4] based on access control lists (ACLs).

Legion represents resources as active objects, which are instances of classes, responsible for their creation and management. Objects communicate via asynchronous method invocations. To facilitate such communication, Legion uses Legion Object Identifiers (LOIDs) for naming. Included in the LOID is the associated object's public key. During Legion object-to-object communication, a recipient authenticates a sender through the use of its well-known LOID and encapsulated public key. Legion can also be configured to support X.509-based authentication.

Legion *credentials* are used to authenticate a user and make access control decisions. When a user authenticates to Legion the user obtains a short-lived, unforgeable credential uniquely identifying himself. A person may possess multiple credentials, signifying multiple roles. When a user invokes an operation of another object, the user's credentials are encrypted and transmitted to the target object. The recipient object unmarshalls the credentials and passes them to a layer (called "MayI") that conceptually encapsulates the object. For each credential passed, MayI determines whether key security critereia are satisfied. Authorization is determined by an ACL associated with each object; an ACL enumerates the operations on an object that are accessible to specific principals (or groups). If the signer of any of these credentials is allowed to perform the operation, the method call is permitted.

Prior to the work described in this paper, the restriction fields of a credential have been unused, because no general-purpose procedure existed by which to determine and implement policy regarding restriction. The credentials have been *bearer* credentials, implying that the possessor or bearer of a particular credential had permission to request *any* operation on behalf of the user. Bearer credentials have been valuable in the development of Legion but are becoming potential security liabilities, especially for Legion's continuing wide-scale deployment. Note that credential interception is not a concern; credentials cannot be intercepted because they are encrypted with the public key of their recipient. Rather, someone could send her credential to an object and that object could then use the credential maliciously in transactions with arbitrary objects. In the next section, we discuss approaches that can be used in general Grids and examine their practical applicability by relating these approaches to Legion.

3 Delegation Approaches

We examine three schemes of restricting delegation: restricting methods that may be called, restricting objects that may pass delegated rights or be the target of such calls, and restricting the validity of delegated rights to a given time period. In each of the following, we start with the base credential: *[The bearer of this credential may do anything on Alice's behalf forever] signed Alice.*

3.1 Method Restrictions

Instead of allowing the presenter of a bearer credential to perform any and all operations, the constructor of a bearer credential can explicitly enumerate allowable methods, either individually or categorically.

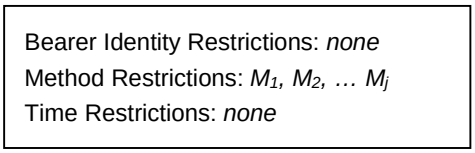

Fig. 1. Bearer credential with method restrictions.

Explicit Method Enumeration by Manual Inspection. A set of well-known object-to-object interactions can be manually identified to determine a remote method invocation tree rooted at a particular remote call, and a credential can list these methods. By restricting the methods that the service can call on her behalf, the user has significantly reduced the ability of an untrustworthy service to perform malicious operations, such as destroy one of her private files.

However, this approach does not require any meaningful context for the credential. This strategy is tedious to implement and inflexible. In large systems, it is extremely difficult, if not impossible, to determine remote method invocations through manual code inspection, e.g., the source code may not even be readily available. Any changes to code shared by many objects (such as Grid infrastructure support code) require that all previously computed restricted credentials be modified to reflect the changes.

Experiences applying explicit enumeration to Legion highlight the limitations of this approach. Many hours were needed to enumerate the method call tree of a simple Legion utility. This utility, like all other objects and utilities, makes use of the services provided by the Legion distributed object framework. While the number of explicit method calls made by the application was small, these calls may lead to a complex web of remote interactions. For example, if the target of a request is not actively running, Legion will need to invoke additional services to schedule and activate the object. These calls were the source of complexity in the exercise and are a likely source of inspection error. We believe that such calls are not specific to Legion, but inherent in any large-scale Grid.

When a user wishes to invoke a remote method call, the user must pass an enumeration containing *each* subsequent remote method invocation. In the cases of large call chains, this list may well be larger than the message itself. This will lead to additional network contention and message setup and transmission latencies. Further, because one monolithic signed enumeration is sent, it cannot be selectively dismantled and will be sent down each branch of the call chain, though many of the enumerated methods will have no bearing on an object or its descendants. The obvious approach is to partition this set into multiple credentials, and generate the

necessary information for a particular object in the call chain to determine *when* it can discard a credential that will not be used in the future.

Compiler Automation. In order for the method enumeration approach to scale beyond a trivial number of operations, the reliance on direct human interaction must be reduced or eliminated. A bottom-up *security compiler* could annotate each remote method call with the remote method calls invoked recursively from it. When compiling an application, the compiler would examine the dependencies to other modules to union these annotations. The compiler would then automatically insert code to initialize the application's credential set in accordance with these annotations. Once the security compiler is written, it is reusable for any operation. In principle, this eliminates many of the limitations of the previous section, particularly the tedious and error-prone nature of generating credentials. If the security compiler is invoked as part of the general compilation process, then this should be viewed as an automatic mechanism with relatively little overhead.

While independence from user intervention removes the tedium of manual enumeration, it also removes the user from the policy decision altogether. That is, the security compiler blindly blesses untrusted code by enumerating the remote method invocations it performs. Under the manual approach, the user is forced to visually inspect the code and may thus discover blatant security violations. For this reason and because of the daunting complexity involved in writing compilers for all of the languages that Legion supports (C, C++, Fortran), this approach has not been pursued for Legion to date.

Method Abstractions. A higher level of abstraction than that of specific method enumeration can be used. As a first approach, methods can be coarsely described as *read* or *write* operations. Then, it is possible to construct a credential for *read* and/or *write* operations in general, in lieu of specific methods. The size and complexity of a credential can thus be significantly reduced. This approach is attractive for high-level object interactions that can be wholly classified as read-only, which seemingly need not authorize write operations (and thus reduce the scope of a compromise).

This approach is limited because it likely does not provide additional security for applications that perform 'write' operations. That is, if a particular remote method invocation entails some destructive or write operation, the credential must grant the whole 'write' capability. Because the Grid infrastructure will almost certainly require some form of 'read' capability (as is the case in Legion), this approach is reduced to an all-inclusive bearer credential. A better approach is to combine the gross categorization of the 'read' capability with specific write operations.

Method abstraction represents a good tradeoff between flexibility and ease of implementation and has been utilized in Legion. Legion was experimentally modified to directly support a 'read' credential by altering the default MayI layer. Next, the 'write' methods made during the execution of a particular tool were enumerated. The all-encompassing 'read' and enumerated 'write' methods form the credential utilized in this tool. Such credentials are currently being tested and evaluated in limited deployment. Unfortunately, such partitioning of the method space requires updating the infrastructure as remote methods are introduced and removed from the system.

3.2 Object Restrictions

In Section 3.1, the scope of attack is limited to those methods enumerated, but the methods are not associated with specific objects and thus may be invoked by *any* object acquiring the credential. By explicitly associating an object or objects with each method, greater security can be provided by restricting not only *what* may be invoked, but also *who* may invoke it or upon *whom* it may be invoked. Object restrictions are discussed as both a complementary and stand-alone measure.

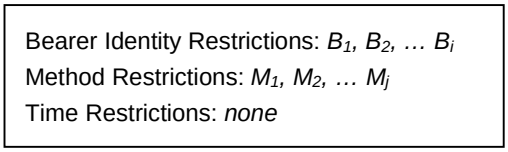

Fig. 2. Bearer credential with method and object restrictions.

Object Classes. Due to the dynamics of the system, it may not be possible for an application to ascertain the identity of all principals involved in a call chain. Furthermore, in some systems, the class or type of the object is more useful than the identity of a particular instance of that class. Therefore, one approach is to restrict the bearer identity to a particular class.

The approach of utilizing class restrictions could follow from the enumeration mechanisms explained above. Unfortunately, it would suffer many of the same setbacks. Nevertheless, when used in tandem with method enumeration, greater security is achieved than via either mechanism in isolation.

This approach is directly applicable to Legion, in which the object hierarchy is rooted at a particular well-known object. All objects are derived directly or indirectly from this class. Legion supports a *classOf* method to describe the relationships throughout the object hierarchy. This approach is currently being considered for a number of Legion operations.

Transitive Trust. A form of transitive trust can be used to address the scalability concern of passing around one all-encompassing credential, as introduced in Section 3.1. In this approach, each remote method invocation is annotated with the methods it invokes directly or those within a certain depth relative to the current position in the call chain. The annotations reflect the target of the method invocation. Unlike previous models discussed, this option would concatenate credentials signed by intermediary nodes throughout the call chain, rather than passing one monolithic list signed by the invoking principal at the root of the call chain.

Embedded metadata within the annotations might allow intermediary nodes to discard credentials that apply neither to themselves nor their descendants. Though the credential set is determined dynamically, the credentials themselves are not created dynamically. They are associated with remote method invocations as before, using either a manual or automatic approach.

While it addresses the scalability concerns of passing bloated credentials, this approach is less secure than the monolithic approaches. A target would recognize an intermediary acting on behalf of the initiator if the credentials form a chain originating at the initiator. While the initiator may specify method restrictions, there is no obvious means for a target to discern that an intermediary is acting in the intended context. For example, two objects could conspire such that the first object gives the second object the ability to execute *arbitrary* methods within the rights of any of its predecessors. To avoid such a coordinated attack, it could be required that the final target object trust each principal named in the chain in order to respect the transitivity properties of that chain. Because of this concern, the use of this approach in Legion is still under consideration.

Trusted Equivalence Classes. Another approach is to group objects into security equivalence classes (e.g. according to importance, functionality, administrative domain, etc.). Instead of specifying an object instance or class name, the name of the equivalence class would be specified as a credential restriction. A practical and important example of such an equivalence class in Legion is the "system" objects. Such objects form the core of the system and include objects representing hosts, schedulers, and storage units. Method invocations initiated from such objects need not be explicitly granted on a per-object basis in the list of rights contained in a credential. This could significantly reduce both the off-line costs and the on-line overhead of generating and sending restricted credentials, as it provides a coarse-grain abstraction. Note that it is not mandated that every user trust a collection of objects, even "system" objects — any user is free to simply not use this approach when constructing credentials.

The major problem in this approach is how to both define these groups and verify membership in these groups, because groups will most likely need to be defined on a per-user basis. Another shortcoming is that a user implicitly trusts any changes made to objects of trusted classes objects.

Trusted Application Writers and Deployers. Rather than trusting classes of objects, trust can be placed in principals that implement those classes. Implicit in this approach is the assumption that an object trusts another object if and only if it trusts the principal which 'vouches' for it. Intuitively, this is more appealing than simply trusting an object which exports a familiar interface but which may have been implemented by a malicious user.

When a user compiles an object or application and makes it available to the general Grid community, the deployment of the object or application in this approach now includes the signature of that user and the signatures attached to any dependent modules — the application is "branded" with the signature of its developer and/or compiler. When a user interacts with services in a Grid, the user sends a credential stating its trust in a specific subset of users and/or compilers. When an object *receives* a request for service, it bases access on the direct or indirect objects that made the request for service. If the request for service on behalf of a particular user is augmented with a credential stating trust in the deployer of the object from which the request arrived, then access is allowed.

In Legion, applications and objects must be *registered* before they can be executed in a distributed environment. Legion manages registered binaries, distributing them to hosts for execution on demand. Registration serves as a convenient time to bind the trust relationship. The principal initiating the registration of a particular binary would be noted as the 'owner' of that object. A user could specify her trust in particular principals or groups. This enumeration could be made explicit in a credential. This approach is appealing because of its generality and the ease with which it allows the instantiation of policy decisions. It should work well for common cases in which a large group of users trust one another (e.g. within a single administrative domain), but do not necessarily trust another group.

3.3 Time-Dependent Restrictions

The last approach is orthogonal to the previous mechanisms and may be used in conjunction with them to restrict the window of vulnerability to replay attacks. In the simplest version of this approach, a pre-determined amount of time is used as the time-out for the credential. This guarantees that the credential cannot be abused in a replay attack after a fixed time interval. Unfortunately, as the perceived slack time in the timeout value is reduced, the likelihood that the credential will need to be refreshed increases. Too much slack time opens the door for a replay attack.

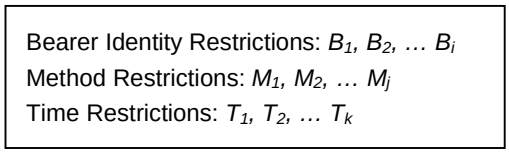

Fig. 3. Bearer credential with method, object, and time restrictions.

If the initiating object will be alive during the entire operation, then it is possible to mandate that each object in the call chain perform a callback to verify that the initiator is indeed still running. This prevents a malicious object from acquiring a bearer credential and using it long after the application has exited but before a fixed timeout value has expired. Unfortunately, this doubles the number of object-to-object transactions and it does not benefit a long-lived application, whose natural execution duration provides plenty of opportunity for a man-in-the-middle to abuse privileges. Callbacks could potentially serve the alternate purpose of recording an audit trail of methods invoked under the identity of a given principal. To limit the performance degradation of indiscriminant callbacks, the principal may require notification of only 'important' method invocations.

This approach has not been used in Legion because it is impossible to predict how long an operation will require, due to the inherent unpredictability in the computational nodes and the networks. Even if this could be performed on a single case, it is not clear to what extent the results can be applied to another tool, as each tool has different functionality. It is also not clear if the increased security of short

time-outs will be of any value to an average user — if more than one operation "fails" in a short time period, the user may think that Legion as a whole is experiencing errors, even though the observed behavior is the result of credential expiration.

4 Related Work

Related delegation work generally assumes that the rights to be delegated have already been decided by a separate mechanism, and it is just a matter of encoding and checking these rights. The general need for delegation is exemplified by the Digital Distributed System Security Architecture, a comprehensive collection of security services and mechanisms for general-purpose distributed systems [7]. The designers of this system note that this form of delegation is unrestricted and that, while restricted delegation "seems desirable", the useful types of restrictions are specific to each application and thus difficult to generalize. Erdos and Pato present an architecture for implementing delegation in the Distributed Computing Environment (DCE) [3]. There are two types of delegation-related restrictions that principals can use to place limitations on revealing their identity – target restrictions and delegate restrictions. Neuman shows how existing authentication systems can be used to support restricted proxies or restricted delegation [10]. In this work, there are two proxy types: a bearer proxy, which may be used by anyone, and a delegate proxy, which may only be used by an object enumerated within. The CORBA Security Specification [2] identifies five types of delegation. The types differ in terms of combining rights, such as whether the initiator's rights are combined or remain distinct from the intermediaries' rights, when a subsequent call must be performed. It is noted that no one security technology supports all options. Linn and Nystrom discuss application of attribute certificates to delegation in distributed environments [9]. Designation of an object's membership in a group is the most common use for attribute certificates. There are numerous ways in which to address delegation with attribute certificates: using a generic attribute certificate as a capability (similar to Legion's bearer credential), enumerating authorized delegates within an attribute certificate, and having an attribute certificate reference a short-term key that could be used by a delegate.

Globus [5] is a Grid toolkit being developed at the Argonne National Lab and USC/ISI. Delegation of rights is a significant problem in Globus (as in Legion), and the Globus developers do not have a solution (although other aspects of the Grid security problem are discussed in [6]). In fact, the reliance on GSS-API [13] in Globus may hinder efforts at delegation in Globus, because GSS-API does not offer a clear solution to delegation (and could impede restricted delegation). There have been some efforts in Globus to adopt and support the Generic Authorization and Access-Control API (GAA API [11]); however, GAA API focuses on mechanism and an API for authorization and does not address a systematic way in which to determine the rights that should be delegated. Independent of Globus, Akenti [12] is a Grid-based project that makes important contributions for a uniform mechanism to perform authorization in a Grid environment but does not directly address the determination of rights to be granted.

5 Conclusions

Computational Grids are a powerful new computing paradigm. Users must currently grant unlimited delegation to services that might act on their behalf. As Grids scale, this implicit trust model will allow potential compromises. Eight general approaches have been described that are valuable for restricting delegation in different situations. To show the utility of these approaches, each was described in the context of Legion, a widely-deployed infrastructure for Grids. Next steps are to determine the characteristic properties of a given situation that determine the most appropriate approach for restricted delegation. The approaches described in this paper are an important first step toward practical restricted delegation, and, as these approaches are further characterized, promise to make using Grids more secure.

References

1. Berman, F., R. Wolski, S. Figueira, J. Schopf, and G. Shao. "Application-level Scheduling on Distributed Heterogeneous Networks", in *Proceedings of Supercomputing 96*, 1996.
2. Chizmadia, David. A Quick Tour of the CORBA Security Service, http://www.omg.org /news/corbasec.htm, Reprinted from Information Security Bulletin-September 1998.
3. Erdos, M.E. and J.N. Pato. "Extending the OSF DCE Authorization System to Support Practical Delegation", *PSRG Workshop of Network and Distributed System Security*, pages 93-100, February 1993.
4. Ferrari, Adam, Frederick Knabe, Marty Humphrey, Steve Chapin, and Andrew Grimshaw. "A Flexible Security System for Metacomputing Environments." In *Seventh International Conference on High Performance Computing and Networking Europe (HPCN Europe 99)*, pages 370-380, April 1999.
5. Foster, Ian, and Carl Kesselman. "Globus: a metacomputing infrastructure toolkit". *International Journal of Supercomputer Applications*, 11(2): pages 115-128, 1997.
6. Foster, Ian, Carl Kesselman, Gene Tsudik, and Steven Tuecke. "A Security Architecture for Computational Grids." In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 83-92, November 1998.
7. Gasser, Morrie, Andy Goldstein , Charlie Kaufman, and Butler Lampson. "The Digital Distributed System Security Architecture." In *Proceedings of 1989 National Computer Security Conference*, 1989.
8. Grimshaw, Andrew S, Adam Ferrari, Frederick Knabe, and Marty Humphrey. "Wide-Area Computing: Resource Sharing on a Large Scale." *Computer*, 32(5): pages 29-37, May 1999.
9. Linn, J. and M. Nystrom. "Attribute Certification: An Enabling Technology for Delegation and Role-Based Controls in Distributed Environments", Proceedings of the *Fourth ACM workshop on Role-Based Access Control*, 1999, pages 121 - 130.
10. Neuman, B. Clifford. "Proxy-Based Authorization and Accounting for Distributed Systems," *Proceedings of the ICDCS'93*, May 1993.
11. Ryutov, T.V., G. Gheorghiu, and B.C. Neuman. "An Authorization Framework for Metacomputing Applications", *Cluster Computing.* Vol 2 (1999), pages 165-175.
12. Thompson, M., W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. "Certificate-based Access Control for Widely Distributed Resources", Proceedings of the *Eighth Usenix Security Symposium*, August 1999.
13. Wray, J. "Generic Security Services Application Programmer Interface (GSS-API), volume 2". RFC 2078, January 1997.