A Security Architecture for Computational Grids*

Ian Foster[1] Carl Kesselman[2] Gene Tsudik[2] Steven Tuecke[1]

[1] Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439
{foster,tuecke}@mcs.anl.gov

[2] Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
{carl,gts}@isi.edu

Abstract

State-of-the-art and emerging scientific applications require fast access to large quantities of data and commensurately fast computational resources. Both resources and data are often distributed in a wide-area network with components administered locally and independently. Computations may involve hundreds of processes that must be able to acquire resources dynamically and communicate efficiently. This paper analyzes the unique security requirements of large-scale distributed (grid) computing and develops a security policy and a corresponding security architecture. An implementation of the architecture within the Globus metacomputing toolkit is discussed.

1 Introduction

Large-scale distributed computing environments, or "computational grids" as they are sometimes termed [4], couple computers, storage systems, and other devices to enable advanced applications such as distributed supercomputing, teleimmersion, computer-enhanced instruments, and distributed data mining [2]. Grid applications are distinguished from traditional client-server applications by their simultaneous use of large numbers of resources, dynamic resource requirements, use of resources from multiple administrative domains, complex communication structures, and stringent performance requirements, among others.

While scalability, performance and heterogeneity are desirable goals for any distributed system, the characteristics of computational grids lead to security problems that are not addressed by existing security technologies for distributed systems. For example, parallel computations that acquire multiple computational resources introduce the need to establish security relationships not simply between a client and a server, but among potentially hundreds of processes that collectively span many administrative domains. Furthermore, the dynamic nature of the grid can make it impossible to establish trust relationships between sites prior to application execution. Finally, the interdomain security solutions used for grids must be able to interoperate with, rather than replace, the diverse intradomain access control technologies inevitably encountered in individual domains.

In this paper, we describe new techniques that overcome many of the cited difficulties. We propose a security policy for grid systems that addresses requirements for single sign-on, interoperability with local policies, and dynamically varying resource requirements. This policy focuses on authentication of users, resources, and processes and supports user-to-resource, resource-to-user, process-to-resource, and process-to-process authentication. We also describe a security architecture and associated protocols that implement this policy. Finally, we present a concrete implementation of this architecture and discuss our experiences deploying this architecture on a large grid testbed spanning a diverse collection of resources at some 20 sites around the world. This implementation is performed in the context of the Globus system [5], which provides a toolkit, testbed, and set of applications that can be used to evaluate our approach. However, we believe that the proposed techniques are general enough to make them applicable outside the Globus context.

In summary, this paper makes four contributions to our understanding of distributed system security:

1. it provides an in-depth analysis of the security problem in computational grid systems and applications;
2. it includes the first detailed formulation of a security policy for grid systems;
3. it proposes solutions to specific technical issues raised by this policy, including local heterogeneity and scalability; and
4. it describes a security architecture that uses these solutions to implement the security policy, and it demonstrates – via large-scale deployment – that this architecture is workable.

2 The Grid Security Problem

We introduce the grid security problem with an example illustrated in Figure 1. This example, although somewhat contrived, captures important elements of real applications, such as those discussed in Chapters 2–5 of [4].

*This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; and by the National Science Foundation.

In proceedings of the 5th ACM Conference on Computer and Communication Security, November 2-5, 1998. San Francisco, CA.

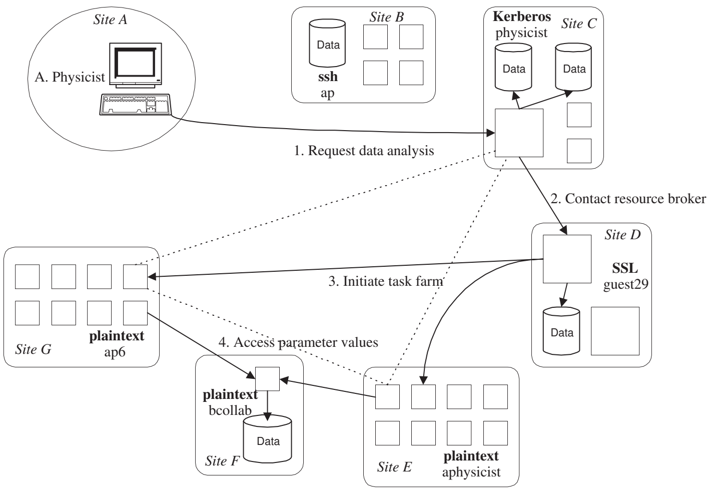

Figure 1: Example of a large-scale distributed computation: user initiates a computation that accesses data and computing resources at multiple locations.

We imagine a scientist, a member of a multi-institutional scientific collaboration, who receives e-mail from a colleague regarding a new data set. He starts an analysis program, which dispatches code to the remote location where the data is stored (site C). Once started, the analysis program determines that it needs to run a simulation in order to compare the experimental results with predictions. Hence, it contacts a resource broker service maintained by the collaboration (at site D), in order to locate idle resources that can be used for the simulation. The resource broker in turn initiates computation on computers at two sites (E and G). These computers access parameter values stored on a file system at yet another site (F) and also communicate among themselves (perhaps using specialized protocols, such as multicast) and with the broker, the original site, and the user.

This example illustrates many of the distinctive characteristics of the grid computing environment:

- The user population is large and dynamic. Participants in such virtual organizations as this scientific collaboration will include members of many institutions and will change frequently.

- The resource pool is large and dynamic. Because individual institutions and users decide whether and when to contribute resources, the quantity and location of available resources can change rapidly.

- A computation (or processes created by a computation) may acquire, start processes on, and release resources dynamically during its execution. Even in our simple example, the computation acquired (and later released) resources at five sites. In other words, throughout its lifetime, a computation is composed of a *dynamic* group of processes running on different resources and sites.

- The processes constituting a computation may communicate by using a variety of mechanisms, including unicast and multicast. While these processes form a single, fully connected logical entity, low-level communication connections (e.g., TCP/IP sockets) may be created and destroyed dynamically during program execution.

- Resources may require different authentication and authorization mechanisms and policies, which we will have limited ability to change. In Figure 1, we indicate this situation by showing the local access control policies that apply at the different sites. These include Kerberos, plaintext passwords, Secure Socket Library (SSL), and secure shell.

- An individual user will be associated with different local name spaces, credentials, or accounts, at different sites, for the purposes of accounting and access control. At some sites, a user may have a regular account ("ap," "physicist," etc.). At others, the user may use a dynamically assigned guest account or simply an account created for the collaboration.

- Resources and users may be located in different countries.

To summarize, the problem we face is providing security solutions that can allow computations, such as the one just described, to coordinate diverse access control policies and to operate securely in heterogeneous environments.

3 Security Requirements

Grid systems and applications may require any or all of the standard security functions, including authentication, access control, integrity, privacy, and nonrepudiation. In this paper, we focus primarily on issues of authentication and access control. Specifically, we seek to (1) provide authentication solutions that allow a user, the processes that comprise a user's computation, and the resources used by those processes, to verify each other's identity; and (2) allow local access control mechanisms to be applied without change, whenever possible. As will be discussed in Section 4, authentication forms the foundation of a security policy that enables diverse local security policies to be integrated into a global framework.

In developing a security architecture that meets these requirements, we also choose to satisfy the following constraints derived from the characteristics of the grid environment and grid applications:

Single sign-on: A user should be able to authenticate once (e.g., when starting a computation) and initiate computations that acquire resources, use resources, release resources, and communicate internally, without further authentication of the user.

Protection of credentials: User credentials (passwords, private keys, etc.) must be protected.

Interoperability with local security solutions: While our security solutions may provide interdomain access mechanisms, access to local resources will typically be determined by a local security policy that is enforced by a local security mechanism. It is impractical to modify every local resource to accommodate interdomain access; instead, one or more entities in a domain (e.g., interdomain security servers) must act as agents of remote clients/users for local resources.

Exportability: We require that the code be (a) exportable and (b) executable in multinational testbeds. In short, the exportability issues mean that our security policy cannot directly or indirectly require the use of bulk encryption.

Uniform credentials/certification infrastructure: Interdomain access requires, at a minimum, a common way of expressing the identity of a *security principal* such as an actual user or a resource. Hence, it is imperative to employ a standard (such as X.509v3) for encoding credentials for security principals.

Support for secure group communication. A computation can comprise a number of processes that will need to coordinate their activities as a group. The composition of a process group can and will change during the lifetime of a computation. Hence, support is needed for secure (in this context, authenticated) communication for dynamic groups. No current security solution supports this feature; even GSS-API has no provisions for group security contexts.

Support for multiple implementations: The security policy should not dictate a specific implementation technology. Rather, it should be possible to implement the security policy with a range of security technologies, based on both public and shared key cryptography.

4 A Grid Security Policy

Before delving into the specifics of a security architecture, it is important to identify the security objectives, the participating entities, and the underlying assumptions. In short, we must define a *security policy*, a set rules that define the security subjects (e.g., users), security objects (e.g., resources) and relationships among them. While many different security policies are possible, we present a specific policy that addresses the issues introduced in the preceding section while reflecting the needs and expectations of applications, users, and resource owners. To our knowledge, the following discussion represents the first such grid security policy that has been defined to this level of detail.

In the following discussion, we use the following terminology from the security literature:

- A *subject* is a participant in a security operation. In grid systems, a subject is generally a user, a process operating on behalf of a user, a resource (such as a computer or a file), or a process acting on behalf of a resource.
- A *credential* is a piece of information that is used to prove the identity of a subject. Passwords and certificates are examples of credentials.
- *Authentication* is the process by which a subject proves its identity to a requestor, typically through the use of a credential. Authentication in which both parties (i.e., the requestor and the requestee) authenticate themselves to one another simultaneously is referred to as *mutual authentication*.
- An *object* is a resource that is being protected by the security policy.
- *Authorization* is the process by which we determine whether a subject is allowed to access or use an object.
- A *trust domain* is a logical, administrative structure within which a single, consistent local security policy holds. Put another way, a trust domain is a collection of both subjects and objects governed by single administration and a single security policy.

With these terms in mind, we define our security policy as follows:

1. The grid environment consists of multiple *trust domains*.

   Comment: This policy element states that the grid security policy must integrate a heterogeneous collection of locally administered users and resources. In general, the grid environment will have limited or no influence over local security policy. Thus, we can neither require that local solutions be replaced, nor are we allowed to override local policy decisions. Consequently, the grid security policy must focus on controlling the interdomain interactions and the mapping of interdomain operations into local security policy.

2. Operations that are confined to a single trust domain are subject to local security policy only.

   Comment: No additional security operations or services are imposed on local operations by the grid security policy. The local security policy can be implemented by a variety of methods, including firewalls, Kerbero,s and SSH.

3. Both global and local subjects exist. For each trust domain, there exists a partial mapping from global to local subjects.

   Comment: In effect, each user of a resource will have two names, a global name and a potentially different local name on each resource. The mapping of a global name to a local name is site-specific. For example, a site might map global user names to: a predefined local name, a dynamically allocated local name, or a single "group" name. The existence of the global subject enables the policy to provide single sign-on.

4. Operations between entities located in different trust domains require mutual authentication.

5. An authenticated global subject mapped into a local subject is assumed to be equivalent to being locally authenticated as that local subject.

   Comment: In other words, within a trust domain, the combination of the grid authentication policy and the local mapping meets the security objective of the host domain.

6. All access control decisions are made locally on the basis of the local subject.

   Comment: This policy element requires that access control decisions remain in the hands of the local system administrators.

7. A program or process is allowed to act on behalf of a user and be delegated a subset of the user's rights.

   Comment: This policy element is necessary to support the execution of long-lived programs that may acquire resources dynamically without additional user interaction. It is also needed to support the creation of processes by other processes.

8. Processes running on behalf of the same subject within the same trust domain may share a single set of credentials.

   Comment: Grid computations may involve hundreds of processes on a single resource. This policy component enables scalability of the security architecture to large-scale parallel applications, by avoiding the need to create a unique credential for each process.

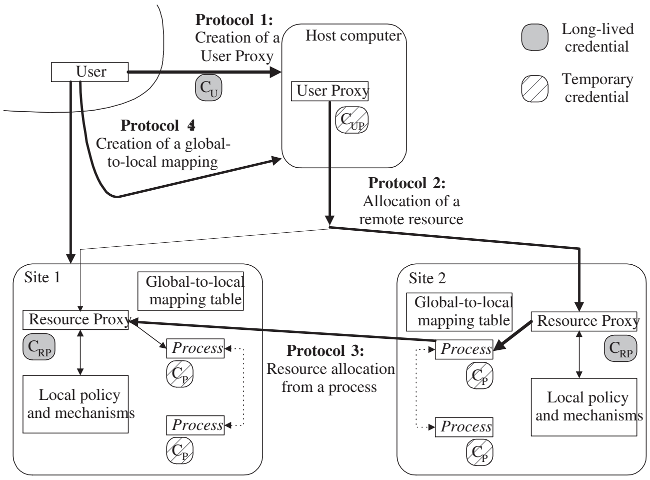

Figure 2: A computational grid security architecture.

We note that the security policy is structured so as not to require bulk privacy (i.e., encryption) for any reason. Export control laws regarding encryption technologies are complex, dynamic and vary from country to country. Consequently, these issues are best avoided as a matter of design. We also observe that the thrust of this policy is to enable the integration of diverse local security policies encountered in a computational grid environment.

5 Grid Security Architecture

The security policy defined in Section 4 provides a context within which we can construct a specific security architecture. In doing so, we specify the set of subjects and objects that will be under the jurisdiction of the security policy and define the protocols that will govern interactions between these subjects and objects. Figure 2 shows an overview of our security architecture. The following components are depicted: entities, credentials, and protocols. The thick lines represent the protocols described later in the paper. The curved line separating the user from the rest of the figure signifies that the user may disconnect once the user proxy has been created; the dashed lines represent authenticated interprocess communication.

We are interested in computational environments. Consequently, the subjects and objects in our architecture must include those entities from which computation is formed. A computation consists of many processes, with each process acting on behalf of a user. Thus, the subjects are users and processes. The objects in the architecture must include the wide range of resources that are available in a grid environment: computers, data repositories, networks, display devices, and so forth.

Grid computations may grow and shrink dynamically, acquiring resources when required to solve a problem and releasing them when they are no longer needed. Each time a computation obtains a resource, it does so on behalf of a particular user. However, it is frequently impractical for that "user" to interact directly with each such resource for the purposes of authentication: the number of resources involved may be large, or, because some applications may run for extended period of time (i.e., days or weeks), the user may wish to allow a computation to operate without intervention. Hence, we introduce the concept of a *user proxy* that can act on a user's behalf without requiring user intervention.

Definition 5.1 *A* user proxy *is a session manager process given permission to act on behalf of a user for a limited period of time.*

The user proxy acts as a stand-in for the user. It has its own credentials, eliminating the need to have the user on-line during a computation and eliminating the need to have the user's credentials available for every security operation. Furthermore, because the lifetime of the proxy is under control of the user and can be limited to the duration of a computation, the consequences of its credentials being compromised are less dire than exposure of the user's credentials.

Within the architecture, we also define an entity that represents a resource, serving as the interface between the grid security architecture and the local security architecture.

Definition 5.2 *A* resource proxy *is an agent used to translate between interdomain security operations and local intradomain mechanisms.*

Given a set of subjects and objects, the architecture is determined by specifying the protocols that are used when subjects and object interact. In defining the protocols, we will use U, R, and P to refer to a user, resource, and process, respectively, while UP and RP will denote a user proxy and resource proxy, respectively. Many of the following protocols will rely on the ability to assert that a piece of data originated from a known source, X, without modification. We know these conditions to be true if the text is "signed" by X. We indicate signature of some text $text$ by a subject X by $Sig_X\{text\}$. This notation is summarized in Table 1.

Table 1: Notation used in the rest of the paper

U, R, P	user, resource, process
UP, RP	user proxy, resource proxy
C_X	credential of subject X
$Sig_X\{text\}$	"text" signed by subject X

The range of interactions that can occur between entities in a computational grid is defined by the functionality of the underlying grid system. However, based on experience and the current grid systems that have been built to date, it is reasonable to assume that the grid system will include the following operations:

- allocation of a resource by a user (i.e., process creation),
- allocation of a resource by a process, and
- communication between processes located in different trust domains.

(We use the term *allocation* to denote the operations required to provide a user with access to a resource. On some systems, this will involve interaction with a scheduler to obtain a reservation [3].) We must define protocols that control UP-RP, P-RP, and P-P interactions. In addition, the introduction of the user proxy means that we must establish how the user and user proxy (U-UP) interact.

Within our architecture, we meet the above requirement by allowing a user to "log on" to the grid system, creating a user proxy using Protocol 1. The user proxy can then allocate resources (and hence create processes) using Protocol 2. Using Protocol 3, a process created can allocate additional resources directly. Finally, Protocol 4 can be used to define a mapping from a global to a local subject.

We now describe each of these protocols in more detail. We note that to minimize problems with export controls, the protocols are all designed to rely on authentication and signature techniques, not encryption. Furthermore, our descriptions do not talk about specific cryptographic methods. In fact, as we shall see below, our implementation uses the Generic Security Services application programming interface to achieve independence from any specific security technology.

5.1 User Proxy Creation Protocol

Recall that a user proxy is an entity within our architecture that acts on behalf of a user. In practice, the user proxy is a special process started by the user which executes on some host local to that user. The main issue in the user proxy creation protocol is the nature of credentials given to the proxy and how the proxy can obtain these credentials.

A user could enable a proxy to act on her behalf by giving the proxy the appropriate credentials (e.g., a password or private key). The proxy could then use those credentials directly. However, this approach has two significant disadvantages: it introduces an increased risk of the credentials being compromised and does not allow us to restrict the time duration for which a proxy can act on the user's behalf. Instead, a temporary credential, C_{UP}, is generated for the user proxy; the user indicates her permission by signing this credential with a secret (e.g., private key). C_{UP} includes the validity interval as well as other restrictions imposed by the user, e.g., host names (where the proxy is allowed to operate from) and target sites (where the proxy is allowed to start processes and/or use resources.)

The actual process of user proxy creation is summarized in Protocol 1. As a consequence of this protocol, the user proxy can use its temporary credential to authenticate with resource proxies.

1. The user gains access to the computer from which the user proxy is to be created, using whatever form of local authentication is placed on that computer.
2. The user produces the user proxy credential, C_{UP}, by using their credential, C_U, to sign a tuple containing the user's id, the name of the local host, the validity interval for C_{UP}, and any other information that will be required by the authentication protocol used to implement the architecture (such as a public key if certificate-based authentication is used):

$$C_{UP} = Sig_U\,\{\textit{user-id, host, start-time, end-time, auth-info}, \ldots\}\,.$$

3. A user proxy process is created and provided with C_{UP}. It is up to the local security policy to protect the integrity of C_{UP} on the computer on which the user proxy is located.

Protocol 1: User proxy creation

The concept of a user proxy is not unique to our architecture. For example, Kerberos generates a limited-lifetime ticket to represent a user. Various public key systems [7, 12], use techniques similar to ours in which temporary credentials (i.e., a public and private key pair) are used to generate a limited lifetime certificate which is then signed by the user to indicate that this certificate represents, or is a proxy for, the user. What distinguishes our architecture from these approaches is the way that a user proxy interacts with the resource proxy to achieve single sign-on and delegation, which is discussed in the next section.

5.2 Resource Allocation Protocol

In discussing resource allocation, we decompose the problem into two classes: allocation of resources by a user proxy and allocation of resources by a process. As process allocation is a generalization of user proxy allocation, we will start our discussion with allocation by a user proxy.

Recall that operations on resources are controlled by an entity, called a *resource proxy*, which is responsible for scheduling access to a resource and for mapping a computation onto that resource. The resource proxy is used as follows. A user proxy requiring access to a resource first determines the identity of the resource proxy for that resource. It then issues a request to the appropriate resource proxy. If the request is successful, the resource is allocated and a process created on that resource. (The procedure would be similar if our goal was simply to allocate a resource, such as network or storage, with which no process was to be associated. However, for brevity, we assume here that process creation always follows resource allocation.)

The request can fail because the requested resource is not available (allocation failure), because the user is not a recognized user of the resource (authentication failure), or because the user is not entitled to use the resource in the requested mode (authorization failure). As discussed above, it is up to the resource proxy to enforce any local authorization requirements. Depending on the nature of the resource and local policy, authorization may be checked at resource allocation time or process creation time, or it may be implicit in authentication and not be checked at all.

We define as Protocol 2 the mechanism used to issue a request to a resource proxy from a user proxy. The verification in Step 3 may require mapping the user's credentials into a local user id or account name if the policy of the resource proxy is to check for authorization at resource allocation time. Alternatively, authorization checks can be delayed until process creation time. The mechanism by which this mapping is performed is discussed in Section 5.4. Notice that the ability to have a resource proxy create credentials on behalf of the process it creates relies on a process and its resource proxy executing in the same trust domain.

The protocol creates a temporary credential for the newly created processes. This credential, C_P, gives the process both the ability to authenticate itself and the identify of the user on whose behalf the process was created. A single resource allocation request may result in the creation of multiple processes on the remote resource. We assign all such processes the same credential, as allowed by security policy element 8. An advantage of this decision is that in the situation when a user allocates resources on large parallel computers, scalability is enhanced. A disadvantage is that it is not possible to use credentials to distinguish two processes started on the same resource by the same allocation request. However, we do not believe that this feature is often useful in practice.

The existence of process credentials enables us to implement a range of additional protocols that allow a process to control access to incoming communication operations on a

1. The user proxy and resource proxy authenticate each other using C_{UP} and C_{RP}. As part of this process, the resource proxy checks to ensure that the user proxy's credentials have not expired.
2. The user proxy presents the resource proxy with a signed request in the form $Sig_{UP}\{allocation specification\}$.
3. The resource proxy checks to see whether the user who signed the proxy's credentials is authorized by local policy to make the allocation request.
4. If the request can be honored, the resource proxy creates a RESOURCE-CREDENTIALS tuple containing the name of the user for whom the resource is being allocated, the resource name, etc.
5. The resource proxy securely passes the RESOURCE-CREDENTIALS to the user proxy. (This is possible from step 1.)
6. The user proxy examines the RESOURCE-CREDENTIALS request, and, if it wishes to approve it, signs the tuple to produce C_P, a credential for the requesting resource.
7. The user proxy securely passes C_P to the resource proxy. (This is again possible due to step 1.)
8. The resource proxy allocates the resource and passes the new process(es) C_P. (The latter transfer relies on the fact that the resource proxy and process are in the same trust domain.)

Protocol 2: Resource allocation (and process creation)

per-subject basis. For example, one can use the process credentials to authenticate a sending process to a destination process, negotiate a session key, and then sign all point-to-point communication, guaranteeing the identity of the sender. The authentication process is simple, since we need simply to check that the other process' credentials are valid, i.e., in the same group.

5.3 Resource Allocation from a Process Protocol

While resource allocation from a user proxy is necessary to start a computation, the more common case is that resource allocation will be initiated dynamically from a process created via a previous resource allocation request. Protocol 3 defines the process by which this can be accomplished.

1. The process and its user proxy authenticate each other using C_P and C_{UP}.
2. The process issues a signed request to its user proxy, with the form

   Sig_P {"allocate", *allocation request parameters* }
3. If the user proxy decides to honor the request, it initiates a resource allocation request to the specified resource proxy using Protocol 2.
4. The resulting process handle is signed by the user proxy and returned to the requesting process.

Protocol 3: Resource allocation from a user process

Admittedly, this technique lacks scalability because of its reliance on a single user proxy to forward the request to the resource proxy. However, this protocol offers the advantage of both simplicity and fine-grained control. While the former is self-evident, fine-grained control requires some elaboration. Consider the obvious alternative of allowing a process (running remotely on behalf of a user) to allocate further resources and create other processes unilaterally. This would have two limitations:

- A user must be able to encode and embed arbitrary policy into each process so as to support individual criteria for resource allocation.
- A security breach or a compromise at a remote site can result in malicious and fraudulent resource allocation purportedly on behalf of an unsuspecting user.

The creation of process specific credentials in protocol 3 results in a *delegation* of a set of rights from the user to the process. The use of delegation for distributed authentication has been addressed in the security literature (e.g., [7]). What sets our approach apart from delegation-based authentication schemes is the role played by the resource proxy. Approaches such as those proposed by [7] require that additional inter-resource trust relationships be established to enable delegation between processes running on those resources. In our protocols, authentication is always between a user proxy and a resource proxy. Consequently, our single sign-on protocol leverages the existing trust relationship between a user and a resource that was established when the user was initially granted access to the resource.

5.4 Mapping Registration Protocol

A central component of the security policy and the resulting architecture is the existence of a "correct" mapping between a global subject and a corresponding local subject. We achieve this conversion from a global name (e.g., a ticket or certificate) into a local name (e.g., login name or user ID) by accessing a *mapping table* maintained by the resource proxy. While a mapping table can be created by the local system administrator, this approach imposes a certain administrative burden and introduces the possibility for error.[1] Hence, we have developed a technique that allows a mapping to be added by a user.

The basic idea behind this technique, presented as Protocol 4, is for a user to prove that he holds credentials for both a global and local subject. This is accomplished by authenticating both globally and directly to the resource using the local authentication method. The user then asserts a mapping between global and local credentials. The assertion is coordinated through the resource proxy, since it is in a position to accept both global and local credentials. In the first two steps, we show the different activities performed by user as it authenticates globally (1.a and 1.b) and to the resource (2.a and 2.b).

Matching MAP-SUBJECT-P and MAP-SUBJECT-UP requests must be issued from both the user proxy and mapping process. This ensures that the same user is in possession of both global and local credentials. If the results of the mapping protocol are stored in a database accessible to the resource proxy, then the user need execute the mapping protocol only once per resource. The duration of time for which a mapping remains valid is determined by local system administration policy. However, we would hope that a mapping will remain in place for the lifetime of either the global credentials or the user's local account.

Part of the mapping protocol requires that the user log into the resource for which the mapping is being created. This requires that a user authenticate themselves to the local system. Consequently, the mapping protocol is only

[1] However, as will be discussed in Section 6.3, some sites actually want to manage the mapping table explicitly as part of their account creation process. Such sites consider protocol 4 as an optional feature.

1.a User proxy authenticates with the resource proxy.

1.b User proxy issues a signed MAP-SUBJECT-UP request to resource proxy, providing as arguments both global and resource subject names.

2.a User logs on to the resource using the resource's authentication method and starts a map registration process.

2.b Map registration process issues MAP-SUBJECT-P request to resource proxy, providing as arguments both global and resource subject names.

1. Resource proxy waits for MAP-SUBJECT-UP and MAP-SUBJECT-P requests with matching arguments.
2. Resource proxy ensures that map registration process belongs to the resource subject specified in the map request.
3. If a match is found, resource proxy sets up a mapping and sends acknowledgments to map registration process and user proxy.
4. If a match is not found within MAP-TIMEOUT, resource proxy purges the outstanding request and sends an acknowledgment to the waiting entity.
5. If acknowledgment is not received within MAP-TIMEOUT, request is considered to have failed.

Protocol 4: Mapping global to local identifier.

as secure as the local authentication method. Clearly, resources with strong authentication (for example based on Kerberos [14], S/KEY, or Secure Shell [22]) will result in a more secure mapping.

6 An Implementation of the Grid Security Architecture

In this section, we describe the Globus Security Infrastructure (GSI), an implementation of our proposed grid security architecture. GSI was developed as part of the Globus project [5], whose focus is to

- understand the basic infrastructure required to support the execution of wide range of computational grid applications,
- build prototype implementations of this infrastructure, and
- evaluate applications on large-scale testbeds.

As part of the Globus project, we have built GUSTO, a testbed that spans over twenty institutions and couples over 2.5 teraflops of peak compute power. This testbed has been used for a range of compute- and communication-intensive application experiments.

As specified by our security architecture, GSI provides support for user proxies, resource proxies (the Globus resource allocation manager (GRAM) [3]), certification authorities, and implementations of the protocols described above. We describe here selected aspects of this implementation, focusing on our use of the Generic Security Services application programming interface (GSS-API), the Secure Socket Layer (SSL), and our experiences deploying the implementation in a large testbed.

6.1 Use of the Generic Security Services Application Programming Interface

The protocols defined above are expressed in terms of abstract security operations, such as signature and authentication, rather than in terms of specific security technologies, such as DES or RSA. Hence, these protocols can be implemented by using any of a number of modern security technologies and mechanisms, such as shared secrets and tickets (e.g., Kerberos), public key cryptography (e.g., SSL), or

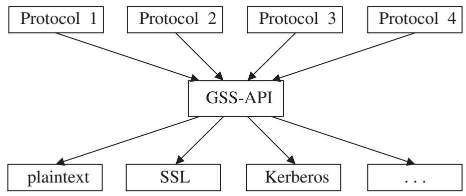

Figure 3: Use of GSS-API in Globus

smart-cards. This separation of protocol and mechanism is a desirable property in an implementation as well, since it enhances the overall portability and flexibility of the resulting system.

To achieve the desired separation, GSI is implemented on top of the Generic Security Services application programming interface (GSS-API) [16]. As the name implies, GSS-API provides security services to callers in a generic fashion. These services can be implemented by a range of underlying mechanisms and technologies, allowing source-level portability of applications to different environments.

GSS-API allows us to construct GSI simply by transcribing the grid security protocols into GSS calls. We can then exploit various grid-level security mechanisms without altering the GSI implementation. The relationship between Globus and GSS-API is shown in Figure 3.

GSS-API is oriented toward two-party security contexts. It provides functions for obtaining credentials, performing authentication, signing messages, and encrypting messages. GSS-API is both transport and mechanism independent. *Transport independence* means that GSS-API does not depend on a specific communication method or library. Rather, each GSS-API call produces a sequence of tokens that can be communicated via any communication method an application may choose. Currently, GSI uses raw TCP sockets and the Nexus communication library [6] to move tokens between processors, although other transports can be easily used as well. *Mechanism independence* means that the GSS does not specify the use of specific security protocols, such as Kerberos, SESAME, DES, or RSA public key cryptography. In fact, a GSS-API implementation may support more than one mechanism and use negotiation-specific mechanisms when the parties in the GSS operation initially contact one another.

GSS-API bindings have been defined for several mechanisms. To date, we have worked with two: one based on plaintext passwords, and one based on X.509 certificates. (In addition, a proof-of-concept Kerberos V5 implementation has been recently completed.) The plaintext password implementation was designed to support system debugging and small-scale deployment, while the certificate-based implementation is used for wide-area "production" use. The flexibility of our GSS-API implementation allows us to switch between public key and plaintext versions of Globus without changing a single line of Globus code.

Remark: While the use of GSS-API has proven to be a significant benefit, the interface is not without limitations. GSS-API does not offer a clear solution to delegation[2], nor does it provide any support for group contexts. The former is needed to allow temporary and limited transfer of user's rights to a process in the event that the user trusts the site

[2] The delegation flag in the *gss_init_sec_context()* notwithstanding.

(and resource) hosting this process enough to forgo an authentication/authorization handshake with the user proxy each time a new process needs to be created. Group context management is needed to support secure communication within a dynamic group of processes belonging to the same computation (or even the same user).

6.2 Support for Public Key Technology in GSI

The GSI implementation currently uses the authentication protocols defined by the Secure Socket Library (SSL) protocol [10]. At first glance, this may seem like an odd choice, since SSL defines a communication layer while GSS explicitly does not. However, in principle, it is possible to separate the authentication and communication components of SSL. To avoid confusion between the SSL authentication protocol and the SSL communication library, we use the term *SSL Authentication Protocol* or SAP to refer specifically to the authentication elements of SSL. We refer to our GSS implementation using SAP as GSS/SAP.

The use of SAP was motivated by several factors. First, there exists a high-quality, public-domain implementation of the SSL protocol (SSLeay), developed outside of the United States and hence avoiding export control issues. Second, SSLeay is structured in a way that allows a token stream to be extracted easily, thus making the GSS implementation straightforward. Third, SSL is widely adopted as the method of choice for authentication and secure communication for a broad range of distributed services, including HTTP servers, Web browsers, and directory services [11]. By combining GSS/SAP with TCP sockets, we can, in fact, reconstitute the entire SSL protocol. Consequently, a computation can use GSI to access not only Globus services, but also generic Web services.

6.3 Deployment

GSI has been deployed in GUSTO, a grid testbed spanning some 20 sites [5] in four countries. GUSTO includes NSF supercomputer centers, DOE laboratories, DoD resource centers, NASA laboratories, universities, and companies.

The initial deployment in late 1997 was limited to the password implementation of GSI and involved installation of the GRAM resource proxy described previously and the establishment of a `globusmap` file that describes the global-to-local mapping applicable at a particular site. Since Protocol 4 above has not yet been implemented, this file is currently maintained manually by site administrators. (We note that, in practice, site administrators often seem to want to maintain this file manually, using it as a form of access control list.) The GRAM resource proxy runs as `root` so as to implement the appropriate mapping for each incoming request.

As mentioned earlier, GSS/SAP is intended to be the default method for Globus applications. After obtaining export approval and license in early 1998, GSS/SAP implementation has been deployed on a wide-scale (both national and international) basis starting in Spring 1998. The password implementation is no longer in production use.

We are also operating a Globus certification authority to support certificate generation for users and resources. To date, resource proxies have been developed that provide gateways to local Kerberos and cleartext/rsh authentication mechanisms.

Our (admittedly limited) experience with GSI deployment offers some confidence that the techniques proposed in this paper are workable. Particularly interesting in this regard is the experience of installing resource proxies at various sites. Because it runs as `root`, resource proxy code was subject to careful review by security administrators at different sites. The result to date has been unanimous approval.

7 Related Work

We distinguish among two main classes of related work: traditional distributed systems security solutions and techniques geared specifically towards large, dynamic, and high-performance computing environments. Not surprisingly, there has been comparatively little work in the latter area.

There are many general-purpose solutions for distributed systems security. Notable examples are Kerberos, DCE, SSH, and SSL. We now review them in brief.

Kerberos has been widely used since the mid-1980s. Although it has evolved considerably during that time, the current MIT release still relies heavily on conventional cryptography and the on-line AS/TGS combination. Recently, optional Kerberos extensions have been proposed to support the use of public key cryptography for certain tasks, including initial user login (PKINIT) [20], interdomain authentication and key distribution (PKCROSS) [19], and peer-to-peer authentication (PKTAPP) [17]. We note that the last two have not progressed past Internet Drafts (expired) and no implementations are available. Although these extensions make Kerberos more attractive (since public key cryptography lends itself to greater security and scalability), Kerberos still remains a fairly heavyweight solution best suited for intradomain security.

DCE is a mature product developed by the Open Group with the security component derived largely from Kerberos. DCE authorization service is much richer and more effective than that of plain Kerberos. In addition to security services, DCE includes a time service, a name service, and a file system. All this is both a blessing and a curse: a blessing since DCE sites get a bundled solution, and a curse, since it is hard to use only selected components of DCE. Furthermore, because of its Kerberos legacy, DCE is based on conventional, shared-key cryptography with trusted third parties (TTPs) such as authentication, ticket granting, authorization, and credential servers. Interdomain security is possible albeit with some complications: on-line presence of TTPs in all domains is assumed. The latest DCE release does support the option of using public keys for initial login. However, the AS/TGS are still assumed to be on line, and public key cryptography is not used for peer-to-peer authentication. Moreover, MIT Kerberos and DCE are not compatible, in particular, where public key use is concerned.

SSH has been developed as a replacement for (mostly UNIX-flavored) remote login, file transfer, and remote execution commands. It is geared primarily for the client-server model. Unlike DCE/Kerberos, SSH is fully public key enabled; that is, all authentication and session key distribution is public key based. SSH supports X.509v3, PGP, and SPKI certificate formats. Also unlike DCE/Kerberos, SSH is oriented toward interdomain communication security. This is a definite plus. However, SSH is essentially an all-or-nothing solution. It provides a secure pipe between the connection end-points and leaves out important elements such as authorization and delegation. SSH does not provide a well-defined API and does not allow decoupling of communication and security services. In addition, SSH's use of bulk encryption is problematic with respect to the overall performance.

SSL is Netscape's secure communication package. It is used primarily for securing HTTP-based Web traffic, although the software is general enough to secure any type of above-transport-layer traffic. SSL supports X.509v3 certificates and uses public key cryptography (RSA) for authentication and key distribution (the latter can be done with either RSA or Diffie-Hellman). Like SSH, SSL is a "secure pipe" solution. Communication and security services are intertwined; SSL assumes a stream-oriented transport layer protocol underneath, for example, TCP. However, we note that SSL allows authenticated, yet nonencrypted, communication.

We now turn to more recent and more specialized solutions aimed at large-scale, wide-area distributed computing.

CRISIS is the security component of Web-OS, an operating system developed for use in wide area distributed computing [21, 1]. Web-OS and Globus are similar in that both aim to provide seamless access to files and computational resources distributed throughout a wide-area network. CRISIS, like GSI, employs SSL for point-to-point secure data transfer and X.509 for certificates.

CRISIS is both a more intrusive and a more complete security architecture. Although it supports local site autonomy insofar as policy, it does not accommodate local security mechanisms. As mentioned earlier, one of our primary goals is to provide a thin layer of homogeneity to tie together disparate and, often incompatible, local security mechanisms. On the other hand, CRISIS encompasses more than just authentication; it also includes extensive access control provisions, caching of credentials, and a secure execution environment, Janus [8].

Unlike Globus, CRISIS does not treat a process as a resource or an entity. This is an important difference because our security architecture allows processes to act independently, for example, to request access to other resources or start another process elsewhere. This makes a running process a temporary principal and, at the same time, a resource jointly owned by the user it belongs to and the local host site.

A further distinction is that we view a grid computation as a dynamic group of peer processes running on different resources in different sites. (Therefore, security in dynamic peer groups is a fundamental issue.) Because of its origins, CRISIS is a more Web-oriented architecture that, although quite suitable for remote execution, is not aimed at (or suitable for) a typical grid computation.

The **Legion** ([9, 15] project also has goals similar to those of Globus, focusing on object-based software technologies for application in grid systems. An object-oriented architecture provides much flexibility with respect to, in particular, security mechanisms. Every object (e.g., file) contains a number of "hooks" allowing security services to be added/extended on a very granular level. However, Legion defines a rather high-level security model without an actual architecture and protocols. In fact, the Globus toolkit can be used to construct an implementation of the Legion's security model.

To summarize, existing distributed computing security technologies are concerned primarily with problems that arise in client-server computing and do not adequately address the issues of creating N-way security contexts, very large (as well as diverse) user and resource sets, or local mechanism/policy heterogeneity.

8 Conclusions and Future Work

We have described a security architecture for large-scale distributed computations. This architecture is immediately useful and, in addition, provides a firm foundation for investigations of more sophisticated mechanisms. We have also described an implementation of this architecture; this implementation has been deployed on a national-scale testbed.

Our architecture and implementation address most of the requirements introduced in Section 3. The introduction of a user proxy addresses the single sign-on requirement and also avoids the need to communicate user credentials. The resource proxy enables interoperability with local security solutions, as the resource proxy can translate between interdomain and intradomain security solutions. Because encryption is not used within the associated protocols, export control issues and hence international use are simplified. Within the implementation, the use of GSS-API provides for portability. Group communication is one major requirement not addressed.

The security design presented addresses a number of scalability issues. The sharing of credentials by processes created by a single resource allocation request means that the establishment of process credentials will not, we expect, be a bottleneck. The fact that all resource allocation requests must pass via the user proxy is a potential bottleneck; this must be evaluated in realistic applications and, if required, addressed in future work. One major scalability issue that is not addressed is the number of users and resources. Clearly, other approaches to the establishment of global to local mappings will be required when the number of users and/or resources are large: on example is the use-condition approaches to authorization [13]. However, we believe the current approach can deal with this.

We hope to develop the techniques described in this paper in four major directions: more flexible policy-based access control mechanisms, based for example on use conditions [13]; representation and implementation of interdomain access control policies; secure group communication, building for example on work in the CLIQUES project [18]; and delegation mechanisms to support scalability to large numbers of resources and users.

Acknowledgments

We gratefully acknowledge Doug Engert's assistance with the development of the SSL implementation of the Globus security architecture, Stuart Martin's contributions to the implementation of the Globus Resource Allocation Manager, and Bill Johnston's comments on a draft of the paper. We also thank the anonymous referees for their insightful critique.

References

[1] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS wide area security architecture. In *Usenix Security Symposium*, January 1998.

[2] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[3] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1998.

[4] I. Foster and C. Kesselman, editors. *Computational Grids: The Future of High Performance Distributed Computing*. Morgan Kaufmann, 1998.

[5] I. Foster and C. Kesselman. The Globus project: A progress report. In *Heterogeneous Computing Workshop*, March 1998.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[7] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *IEEE Symposium on Research in Security and Privacy*, pages 20–30, May 1990.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications — confining the wily hacker. In *Proc. 1996 USENIX Security Symposium*, 1996.

[9] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 1994.

[10] K. Hickman and T. Elgamal. The SSL protocol. Internet draft, Netscape Communications Corp., June 1995. Version 3.0.

[11] T. Howes and M. Smith. A scalable, deployable directory service framework for the internet. Technical Report CITI TR-95-7, CITI, University of Michigan, July 1995.

[12] D. Hühnlein. Credential management and secure single login for SPKM. In *ISOC Network and Distributed System Security Symposium*, March 1998.

[13] W. Johnston and C. Larsen. A use-condition centered approach to authenticated global capabilities: Security architectures for large-scale distributed collaboratory environments. Technical Report 3885, LBNL, 1996.

[14] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). Internet RFC 1510, September 1993.

[15] M. Lewis and A. Grimshaw. The core Legion object model. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 562–571. IEEE Computer Society Press, 1996.

[16] J. Linn. Generic security service application program interface, version 2. Internet RFC 2078, January 1997.

[17] A. Medvinsky and M. Hur. Public key utilizing tickets for application servers. Internet draft, January 1997.

[18] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A new approach to group key agreement. In *IEEE ICDCS'98*, May 1998.

[19] B. Tung, T. Ryutov, C. Neuman, G. Tsudik, B. Sommerfeld, A. Medvinsky, and M. Hur. Public key cryptography for cross-realm authentication in Kerberos. Internet draft, November 1997.

[20] B. Tung, J. Wray, A. Medvinsky, M. Hur, and J. Trostle. Public key cryptography for initial authentication in Kerberos. Internet draft, November 1997.

[21] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating system services for wide area applications. Technical Report UCB CSD-97-938, U.C. Berkeley, 1997.

[22] T. Ylonen, T. Kivinen, and M. Saarinen. SSH protocol architecture. Internet draft, November 1997.