

# A Flexible Security System for Metacomputing Environments\*

Adam Ferrari, Frederick Knabe, Marty Humphrey,  
Steve Chapin, and Andrew Grimshaw

Department of Computer Science  
University of Virginia, Charlottesville, VA 22903, USA

Technical Report CS-98-36

December 1, 1998

## Abstract

A metacomputing environment is a collection of geographically distributed resources (people, computers, devices, databases) connected by one or more high-speed networks and potentially spanning multiple administrative domains. Security is an essential part of metasystem design—high-level resources and services defined by the metacomputer must be protected from one another and from possibly corrupted underlying resources, while those underlying resources must minimize their vulnerability to attacks from the metacomputer level. We present the Legion security architecture, a flexible, adaptable framework for solving the metacomputing security problem. We demonstrate that this framework is flexible enough to implement a wide range of security mechanisms and high-level policies.

## 1 Introduction

Legion [5, 6] is a distributed computing platform for combining very large collections of independently administered machines into single, coherent environments. Like a traditional operating system, Legion builds on a diverse set of lower-level resources to provide convenient user abstractions, services, and policy enforcement

---

\*This work was funded by DARPA contract N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, and DOE D459000-16-3C

mechanisms. The difference is that in Legion, the lower-level resources may consist of thousands of heterogeneous processors, storage systems, databases, legacy codes, and user objects, all distributed over wide-area networks spanning multiple administrative domains. Legion provides the means to pull these scattered components together into a single, object-based *metacomputer* that accommodates high degrees of flexibility and site autonomy.

Security is an essential part of the Legion design. In a metacomputing environment, the security problem can be divided into two main concerns:

1. Protecting the metacomputer's high-level resources, services, and users from each other and from possibly corrupted underlying resources, and
2. Preserving the security policies of the underlying resources that form the foundation of the metacomputer and minimizing their vulnerability to attacks from the metacomputer level.

For example, restricting who is able to configure a metacomputer-wide scheduling service would fall in the first category, and its solution requires metacomputer-specific definitions of identity, authorization, and access control. Meanwhile, enforcing a policy that permits only those metacomputer users who have local accounts to run jobs on a given host falls in the second category, and it might require a means to map between local identities and metacomputer identities.

To satisfy users and administrators, a full security solution must address and reconcile both of these security concerns. Users must have confidence that the data and computations they create within the metacomputer are adequately protected. Administrators need assurances that by adding their resources to a metacomputer (and thus making those resources more accessible and valuable to users), they are not also introducing unreasonable security vulnerabilities into their systems.

Attempting to incorporate security as an add-on late in the implementation process has been problematic in a number of first-generation metacomputing systems such as PVM, MPI, and Mentat. To avoid this pitfall, the Legion group has addressed security issues since the earliest design phases [13]. Our metacomputing security model has three interrelated design goals:

**Flexibility.** The framework must be adaptable to many different security policies and allow multiple policies to coexist.

**Autonomy.** Organizations and users within a metacomputing environment should be able to select and enforce their desired security policies independently.

**Breadth.** The metacomputer's architectural framework must enable a rich set of security policy features.

These goals are strongly driven by our view that a fundamental capability of a metacomputer is its ability to scale over and across multiple trust domains. A Legion “system” is really a federation of meta- and lower-level resources from multiple domains, each with its own separately evaluated and enforced security policies. As such, there is no central kernel or trusted code base that can monitor and control all interactions between users and resources. Nor is there the concept of a superuser—no one person or entity controls all of the resources in a Legion system.

If it is to satisfy a broad range of security needs, our architecture must allow the implementation of a number of different security features. These include:

**Isolation.** Components in the metacomputer should be able to insulate themselves from security breaches in other parts of the system. This feature is particularly important in large-scale systems, where we must generally assume that at least some of the underlying hosts have been compromised or may even be malicious.

**Access control.** Resources typically require access control mechanisms that embody authentication and authorization policies.

**Identity.** The ability to assert and confirm identity is essential for access control, nonrepudiation, and other basic functions.

**Detection and recovery.** A metacomputing environment should support mechanisms for detecting intrusion and misuse of resources, and for recovering after a security breach.

**Communication privacy and integrity.** Communication over the networks that bind the metacomputer together may need to be encrypted or protected if the networks cannot themselves be trusted.

**Standards.** Existing security standards such as Kerberos, ssh, DCE, etc., may need to be integrated into the metacomputing environment to satisfy local administrative policy and to handle legacy software.

In this paper we elaborate a metacomputing architecture based on our design goals that addresses both parts of the metacomputing security problem. We also describe a wide set of mechanisms we have designed or implemented that enable a number of useful security policies, and provide examples of those policies. The key strength of our framework is its ability to support these policies and mechanisms along with many others.

The rest of the paper is organized as follows: Section 2 describes the Legion system architecture, concentrating on the elements most closely related to security in the system. Section 3 describes concrete security mechanism designs that we have implemented within the framework of the Legion architecture. Section 4 discusses examples of high-level policies, and how these policies can be implemented within the Legion security system. Section 5 describes related systems and approaches. Section 6 contains conclusions.

## 2 Architecture

Legion was designed with the explicit intent of supporting a powerful, flexible security architecture. Basic Legion design principles such as encapsulation, extensibility, flexibility, autonomy, and scalability have resulted in a system that can support the varied requirements of application programmers, tool builders, and resource providers. In this section, we examine the fundamental elements of the Legion architecture, introducing the concepts that will be the basis for the rest of the paper.

### 2.1 Object Model

Legion is composed of independent, active objects; all entities of interest within the system—processing resources, storage, users, etc.—are represented by objects [7]. Legion objects communicate via asynchronous method invocations supported by an underlying message passing system. Each method invocation contains a set of explicit (i.e., actual) parameters, and an optional set of arbitrary *implicit* parameters, attribute–value pairs that are available to called objects as invocation metadata. Method calls can produce an arbitrary set of results. Using data-flow information encoded in Legion method invocations, results are forwarded directly to where they are needed, rather than necessarily back to the caller. Objects are instances of classes that define their interface, which is required to be a superset of a minimal *object-mandatory* interface. Object mandatory methods include functions such as an interface query and methods to implement object persistence.

Legion objects are persistent and are defined to be in one of two states: *active* or *inert*. When an object is active, it is hosted within a running process and can service method invocations. When an object is inert, its state (called its Object Persistent Representation, or OPR) is stored on a persistent storage device managed within the system. Objects implement internal methods to store and recover their dynamic state.

For the purpose of communication, every object is identified by a unique,

location-independent Legion Object Identifier, or LOID. LOIDs consist of a variable number of variable length binary fields. Some LOID fields are reserved for system-level identification purposes, e.g., one field is used to identify the object's class, and another contains an instance number for the object (unique within the object's class). Additional fields can be used to contain other information about the object, for example, location hints or security information such as the object's public key.

## 2.2 Core Objects

Within this general object model, Legion defines the interfaces to a set of basic classes that are fundamental to the operation of the system, and that support the implementation of the object model itself. The most important core Legion object classes for the purposes of this discussion are Host Objects, Vault Objects, and Class Manager objects.<sup>1</sup>

Host Objects in Legion represent processing resources. When a Legion object is activated, it is a Host Object that actually creates a process to contain the newly activated object. The Host Object thus controls access to its processing resource and can enforce local policies, e.g., ensuring that a user does not consume more processing time than allotted.

Vault Objects in Legion represent stable storage available within the system for containing OPRs. Just as Host Objects are the managers of active Legion objects, Vault Objects are the managers of inert Legion objects. For example, Vaults are the point of access control to storage resources, and can enforce policies such as file system allocations.

Hosts and Vaults provide the system with interfaces to processing and storage resources. The use of these interfaces is encapsulated by Class Manager Objects. Class Managers are responsible for managing the placement, activation, and deactivation of a set of objects of a given class. They provide a central mechanism for specifying policy for a set of like objects. Policies set by the Class Manager include defining which implementations are valid for instances, which hosts are suitable for execution of instances, which users may create new instances, and so on.

In addition to setting policy for instances, Class Managers serve as location authorities for instances. To accomplish message passing in Legion, LOIDs must be bound to low-level object addresses (typically an IP address plus port number). This binding process is supported by Class Managers, which maintain a record of each instance's object address. The binding process also serves as an automatic

---

<sup>1</sup>In many of the cited Legion references, Class Manager Objects are referred to simply as "Class Objects."

object activation mechanism in Legion: if a binding request for an inactive instance is received, the Class Manager automatically activates the referred-to object so it can service the pending method. The Legion message system defines a rebinding mechanism that is automatically invoked when bindings become stale, for example due to object migration.

Class Managers are first-class objects themselves, and are thus managed by higher-level Class Managers known as Meta-Class Managers. Meta-Class Managers are in turn managed by yet higher-level managers and so on. The recursion for a given Legion domain halts at a logically central Class Manager known as a Legion Class Manager. Within a domain, the location of the Legion Class Manager is well-known to ensure that the recursive binding process will terminate. A complete Legion system can be composed of any number of such hierarchies; inter-domain binding traffic is automatically forwarded among the cooperating Legion Class Managers, as depicted in Figure 1.

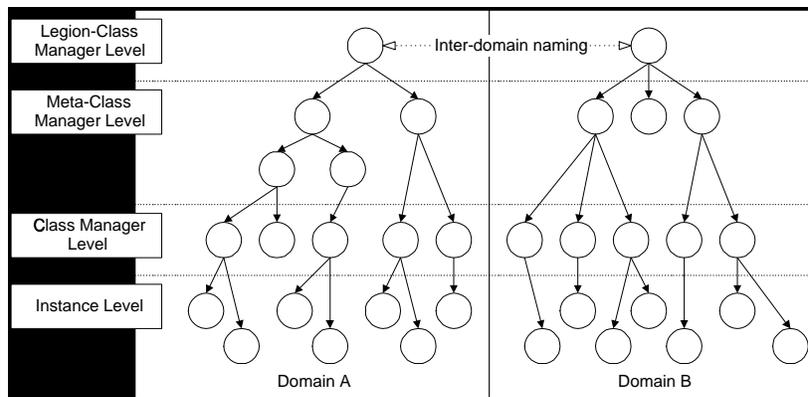


Figure 1: Legion Class Manager hierarchies depicted for a system consisting of two Legion domains. Solid arrows represent the “instance of” relationship.

A critical aspect of the Legion core object classes is that they define interfaces, not implementations. The Legion software distribution provides a number of default reference implementations of each core object type, but the model explicitly enables and encourages the configuration, extension, and even replacement of local core object implementations to suit site- and user-specific functionality and performance requirements. For example, by replacing the implementation of the Host Object, a site can define arbitrary mechanisms and policies for the usage of their computational resources.

### 2.3 Legion Runtime Library

The implementation of Legion objects, including the core object types, is supported by a *Legion Runtime Library* (LRTL) interface. The LRTL provides services analogous to a traditional Object Request Broker (ORB), defining the interfaces to services such as message passing, object control (e.g., creation, location, deletion), dynamic invocation construction, distributed exception handling, and other basic required mechanisms.

A critical element of the LRTL is its flexible, configurable protocol stack [12]. All of the processing performed during the construction and execution of invocations on the object-caller side, and in the receipt, assembly, and service of invocations on the object-implementation side, is configured using a flexible event-based model. This feature of the Legion software allows tool builders to provide drop-in protocol layers for Legion object implementations in a convenient fashion. For example, adding message privacy through a cryptographic protocol is simply a matter of registering the appropriate message processing event handlers into the Legion protocol stack—the added service is transparent to the application developer.

### 2.4 Security Principles

Thus far, the presented architecture has specified neither security mechanisms nor policies. This loose specification is intentional—Legion is intended to provide a framework suitable for implementing the widest possible range of application-level and resource-level security mechanisms and policies, without dictating any globally required mechanisms or policies. It is our goal to enable sites to expose their resources to Legion in a manner compliant with their local policies. Similarly, it is our goal to enable application programmers to achieve desired, application-specific trade-offs among type, level, and cost of security policies and mechanisms.

This said, it is also our goal to provide practical metacomputing software that is extremely easy to use in the common case, both for system administrators and application developers. Towards this end, we have designed and implemented a number of fundamental security mechanisms within the Legion framework. We also embed a number of simple, conservative default policies in the system software that we expect will be useful in the common case. The default policies can always be overridden with ease, and thus do not detract from the flexibility of the system. However, in our experience, many users are only willing to execute software “out of the box” with minimal configuration effort. We want Legion to support these users safely and effectively as well.

We note that the description of the Legion architecture provided in this section is only a quick overview of the elements necessary for a discussion of Legion secu-

rity. Complete details of the Legion architecture and implementation are described in other sources, e.g., [4, 7].

### **3 Implementation and Default Policies**

In Section 2 we described the basic Legion system architecture. Within this architecture we have designed and implemented a set of basic security mechanisms and policies. In this section we describe these basic, default Legion security mechanisms, concentrating on the types of security properties these mechanisms enable within the abstract Legion architecture.

#### **3.1 Identity**

Identity is fundamental to higher-level security services such as access control. In Legion, identity can be based most naturally on LOIDs, since all entities of interest (including users) are represented as Legion objects. As a default Legion security practice, we use one of the LOID fields to contain security information including an RSA public key.

By including the public key in an object's LOID, we make it easy for other objects to encrypt communications to that object or to verify messages signed by it. Objects can just extract the key from the LOID, rather than looking it up in some separate database. By making the key an integral part of an object's name, we eliminate some kinds of public key tampering. An attacker cannot substitute a new key in a known object's id, because if any part of the LOID is altered, including the key, a new LOID is created that will not be recognized by Class Manager Objects during the binding process, and so on. One drawback, though, is that there is no mechanism for revoking an object's key and issuing a new one, as this step implies a complete change of the object's name.

An object normally gets its LOID from its Class Manager when it is created. The Class Manager assigns a new instance number to the object and creates its keys. The resulting LOID and keys are communicated over an encrypted channel to the Host Object on the machine where the object will run. Once the object is up, the Host Object passes its LOID and keys to it over a temporary socket connection. With a LOID in its possession, the object can now begin communicating with other objects using normal Legion mechanisms.

Certain objects are not created by Host Objects and get their LOIDs in different ways. Command-line Legion programs create their LOIDs and keys themselves. The instance number is chosen at random, and the new LOID is registered with a special command-line Class Manager for the domain. The private key is never

transmitted. Another special case is the core system objects that are necessary to bootstrap a Legion domain. These have their LOIDs and keys generated by a special domain initialization program.

Users also have LOIDs. A user creates his own LOID, which is then registered with a user Class Manager and entered in appropriate system groups and access control lists by the respective administrators. When an object such as a command-line program calls another object on behalf of the user, the user's LOID and associated credentials provide the basis for authentication and authorization. The ownership of a LOID resides in the user's unique knowledge of the private key that is paired with his LOID. The private key is kept encrypted on disk, on a smart card, or in some other safe place.

Although LOIDs serve as ids in Legion, they are not easily manipulated by people. The same service objects in different Legion systems will have different LOIDs, making it hard to write utility programs based on raw LOIDs. To solve these problems, Legion provides a directory service called *context space* that maps string names to LOIDs. A context contains string entries which may be linked to any kind of Legion object. Objects in a context may be files, hosts, users, etc., as well as other contexts. Context space is similar to a Unix file system; the contexts resemble directories where all the entries are soft links.

Contexts make it much easier to identify services and objects in a Legion system. However, this convenience also introduces some risks. Once objects rely on contexts to look up services, the focus for an attacker becomes the contexts themselves. If an attacker compromises a context, he can replace the LOIDs of valid objects with LOIDs of his own. There is nothing new about this type of vulnerability (an attacker who gains root access on Unix can easily replace system programs, for example), but it points out that LOIDs and their integral public keys do not protect against spoofing.

### **3.1.1 X.509 Certificates**

The default security field of a LOID is more than just an RSA public key. It is actually an X.509 certificate that contains the key. In general, an X.509 certificate pairs a public key with a person's name, organization, identification of the public key algorithm, and other information. A certificate may be signed by a certification authority (CA) that vouches for the association of the key with the identifying information. To cover the case where a recipient doesn't recognize the CA, the CA's own certificate can be chained onto the certificate, allowing the CA's CA to be the basis of authority. The chain can have multiple links, each link generally leading to a higher authority CA. A recipient validates a certificate by traversing the chain until it reaches a CA it recognizes (for example, Verisign or the U.S.

Postal Service) and checks that all the intervening certificates are properly signed. Validators are free to put constraints on how deep a chain they will accept, who they will or will not trust, etc.

By default, each user in a Legion system has a signed X.509 certificate. If an organization installing Legion does not currently use X.509 certificates or endorse a particular CA, the Legion administrator can set up his own certification authority for Legion users. Some CA named in the CA chain on user certificates should be recognized by all the potential validators in a Legion system, but this is not a requirement; objects can have their own policies for handling method calls that include certificates they can't validate.

The user's X.509 certificate is propagated with requests and method calls made directly or indirectly on behalf of the user. The information in the certificate is used when making entries to access logs. It can also be the basis for alternative access control mechanisms, which we will discuss later.

All normal Legion objects also have X.509 certificates containing their public keys. However, the name fields of these certificates are empty, and the certificates are left unsigned. The main use of X.509 for Legion objects is to encode public keys in a standard way.

### **3.1.2 Credentials**

For a resource, the essential step in deciding whether to grant an access request is to determine the identity of the caller. If a user communicates directly with the target object, he can establish his identity relatively easily with an authentication protocol, which typically involves performing an operation that only someone in possession of his private key can do. In a distributed object system, however, the user typically accesses resources indirectly, and objects need to be able to perform actions on his behalf (for example, a user does not invoke the services of a Host Object directly, but instead relies on a Class Manager to use Host services). Though intermediate objects could in principle be given the user's private key, the risk involved is too great. Given the user's private key, an object can do anything the user can. That's more privilege than is usually necessary.

To avoid the need for sharing the private key, we could have resources call back to the user or his trusted proxy when they receive access requests in the user's name. This step puts control back in the user's hands. There are several drawbacks to this approach, though. First, the fine-grain control afforded by authorization callbacks may be mostly illusory. It can be very difficult to craft policies for a user proxy (or even the real user himself!) that are much more than "grant all requests"—too much contextual and semantic information is generally missing from the request. Beyond this barrier, callbacks are expensive and do not scale well. Though there

is nothing in the Legion architecture that precludes using callbacks for particular objects or resources (and in some cases they may indeed be appropriate), calling back for authorization is not a universal solution. In Legion, after all, every object represents a resource of some type, and a callback on every method call would be a crippling performance hit.

The intermediate solution between these approaches is to issue *credentials* to objects. A credential is a list of rights granted by the credential's maker, presumably the user. They can be passed through call chains. When an object requests a resource, it presents the credential to gain access. The resource checks the rights in the credential and who the maker is, and uses that information in deciding to grant access.

There are two main types of credentials in Legion: *delegated credentials* and *bearer credentials*. A delegated credential specifies exactly who is granted the listed rights, whereas simple possession of a bearer credential grants the rights listed within it. A Legion credential specifies

- The period the credential is valid
- Who is allowed to use the credential
- The rights—which methods may be called on which specific objects or class of objects

If missing, fields default to “all.” The credential also includes the identity of its maker, who digitally signs the complete credential.

A sample delegated credential is “[*Object A may call object B's method M as Alice during the period T*] signed Alice.” To use this credential, A must authenticate to B when it makes its request. We don't have to worry about protecting the credential from theft, because only A can use it. Moreover, the specification of the target object, the method to be called, and the timeout closely limit how this credential can be used. Greater specificity lowers the risk of giving away rights that can be misused by other parties.

It is not always possible to specify a credential so narrowly. Call chains can be long, and the identity of the final object making a resource request may be unknown. If the call chain branches out, several different objects from different classes may need to make calls on the user's behalf. We can loosen the specificity of credentials to handle these cases, but risk increases at the same time. The credential “[*The bearer has all of Alice's rights forever*] signed Alice” is very convenient to give to objects, as there is no danger of accidentally restricting any actions, but should the credential be stolen, Alice is in trouble.

In Legion, tools or commands directly executed by the user create the credentials they need to carry out their actions. The credentials are made as specific as

possible. For example, if the user executes the command to create an object instance, that command will create a credential that authorizes the specified Class Manager to create an object instance on a Host Object. Of course, the Host Object that is contacted might reject the object creation request, but this rejection would be because the user was not authorized to use that resource, not because the Class Manager lacked the authorization to act on the user's behalf.

If a long-lived bearer credential is known to be stolen, the recovery strategy is to create a new LOID for the user. The new LOID has a different instance number but reuses the user's official X.509 certificate for its security field. The resource providers must then modify the appropriate system groups and replace the user's old LOID with the new one. The user's old objects will need new credentials to access resources, and we will see shortly how they can get those. The flaw, though, is in having long-lived bearer credentials to begin with.

### 3.1.3 Credential Refresh

The primary reason for limiting the duration of credentials, particularly bearer credentials, is to limit the period during which a credential is vulnerable to theft or abuse. If credentials expire too rapidly, however, valid objects will not be able to use them for their intended purposes. Class objects in particular must be able to hold credentials for long periods of time, so that the objects they manage can be reactivated without user intervention.

In Legion, we establish a balance between these conflicting goals. By default, we make credentials expire relatively quickly (for example, after some portion of an hour). A holder of an expired credential can get a fresh one by contacting a special *Credential Refresh Object* owned by the user. The Refresh Object then hands back an equivalent, fresh credential.

The purpose of the Refresh Object is to provide a single point of policy for handling credential revocation. If a security breach is suspected, there are a number of possibilities:

- The Refresh Object can stop renewing credentials issued before the security breach was discovered, so that all prior credentials (including those held by the attacker) will time out relatively quickly.
- The Refresh Object can log information about refresh requesters, or require authentication information.
- The Refresh Object can grant refreshes on some kinds of credentials but not others. For example, general bearer credentials may not be refreshable, or only refreshable once.

A Refresh Object needs the ability to sign credentials on behalf of the user. However, we don't want to give it the user's private X.509 key. That key should be treated very carefully, because handling its loss is likely to be expensive—the user's electronic identity would need to be revoked and reissued for all applications he uses it for. Normally the user's private key will be kept in a form that requires the user's direct participation to access. For example, it may be encrypted on disk or stored on a smart card.

Instead of using the X.509 private key, we generate a special public key pair when setting up a new Legion user for the first time. With his X.509 private key, the user then creates and signs a special *proxy credential*. The proxy credential specifies that regular credentials created with the new key pair are equivalent to credentials directly created by the user and his X.509 key. We call this key pair the *proxy keys*.

A Refresh Object is initially configured with the user's proxy keys and proxy credential. When the Refresh Object receives a request for a fresh credential, it generates a new one using the proxy keys. The new credential is sent back along with the proxy credential. A resource validates a request by checking both of the credentials together.

Proxy credentials are long-lived; if they expired quickly, Refresh Objects would periodically need new proxy credentials from a potentially absent user. A proxy credential alone is worthless, however, and the normal credentials they accompany do time out, so the long life of proxy credentials is not a security problem.

If a user's proxy keys are stolen, the attacker can easily create new credentials in the user's name. Recovery is the same as if a long-lived bearer credential had been stolen: The user must change his LOID. To reduce the risk of theft, the Refresh Object should be assigned to a Host Object and Vault Object trusted by the user and less likely to be compromised.

The name of a user's Refresh Object is sent along as an implicit parameter in method calls made on the object's behalf. The Refresh Object is essential for objects that may need to hold a user's credential for a long period of time. For example, a Class Manager may need to reactivate a user's object multiple times during the object's lifetime, which may be months or even years. The Refresh Object allows the Class Manager to get the credentials it needs even if the user is not available.

Though in normal use calls to the Refresh Object should be at a relatively low rate, it is possible that it could become a hot spot and affect scalability. There is nothing that prevents a user from having multiple Refresh Objects if load becomes a problem. However, there is an increased level of risk in having more outstanding copies of proxy keys. One mitigation strategy may be to have multiple sets of proxy keys and to break Refresh Objects into different trust categories. Less

trusted Refresh Objects might be given proxy keys and credentials that only allow refreshing of delegated credentials, not bearer credentials.

#### **3.1.4 Command-Line Credentials**

Command-line Legion programs also generate credentials on behalf of the user. However, we do not want them to use the user's X.509 private key. Even if the commands are interactive and we can assume the user is present, the X.509 private key is too inconvenient to use. For example, we can't have every Legion command prompt for a pass phrase so that it can decrypt the private key.

Just as with Refresh Objects, we use proxy keys and proxy credentials for command-line objects. The private proxy key is stored in the clear on disk, though protected by the local file system. The proxy credential is also available on disk so that commands may send it with the new credentials they generate. Simply by gaining access to his files, the user is "logged on" to Legion.

Another mechanism is also available if storing proxy keys on disk is considered too risky. To start using Legion, the user runs a special login tool that generates new proxy keys and credentials (and thus temporarily requires access to the user's X.509 private key). This login tool stores the keys in memory and creates a new subshell for executing Legion commands. The commands use a private socket to obtain the proxy keys from the parent login program. When the subshell is exited (i.e., the user "logs out"), the keys and credentials are discarded, though they may live on in the user's Refresh Object.

#### **3.1.5 Authentication Credentials**

To use a delegated credential that it holds, an object needs to authenticate itself to the target object. It does this by sending an *authentication credential* with the call. The authentication credential is the LOID of the target and is signed by the object. It is sent directly to the target and protected from theft en route. By "protected from theft" we mean that it cannot be extracted and combined with any other method call (how the communication layer handles this is discussed in Section 3.3). This protection is necessary because anybody holding both the authentication credential and the delegated credential can access the target, and the delegated credential may be known by many other parties.

By making the authentication credential hold the target's LOID, we ensure that the target object itself cannot misuse the authentication credential. Otherwise, the target might use the authentication credential along with some other credential delegated to the object to gain access to another resource.

### 3.1.6 Key Sharing

Public key pairs are expensive to generate. This expense is particularly noticeable for interactive commands, which have to create LOIDs for themselves, and for programs that create large collections of slave objects to carry out parallel computations.

We can largely eliminate this expense by sharing key pairs over collections of objects. The LOIDs are still distinct because they have different instance numbers, but the security fields are the same. In the case of command-line programs, we can simply use the proxy keys for the command-line objects' private keys and LOIDs. For other objects such as MPI slaves, the responsible Class Manager can generate one public key pair and use it for all the objects it creates.

The drawback of sharing keys is that if the private key of one object is stolen, all of its partners are immediately compromised as well. By only sharing a key within applications or for an interactive session, this risk is reduced. Otherwise, Class Managers can continue to generate new keys for each new object they create, and can improve performance by pregenerating a pool of keys when idle.

## 3.2 Access Control

In Legion, access is the ability to call a method on an object. The object may represent a file, a Legion service, a device, or any other resource. Access control is not centralized in any one part of the Legion system. Each object is responsible for enforcing its own access control policy. It *may* collaborate with other objects in making an access decision, and indeed, this allows an administrator to control policy for multiple objects from one point. The Legion architecture does not require this, however.

The general model for access control is that each method call received at an object passes through a *MayI* layer before being serviced. MayI is specified as an event in the configurable Legion protocol stack [12]. MayI decides whether to grant access according to whatever policy it implements. If access is denied, the object will respond with an appropriate security exception, which the caller can handle any way it sees fit.

MayI can be implemented in multiple ways. The trivial MayI layer could just allow all access. The default LRTL implementation provides a more sophisticated MayI that implements access control lists and credential checking. In this MayI, access control lists can be specified for each method in an object. There are two lists for each method, an *allow* and a *deny*. The entries in the lists are the LOIDs of callers that are granted or denied the right to call the particular method; a deny entry supersedes an allow. Default allow and deny lists can be specified to cover

methods that don't have their own entries.

The LOIDs in the allow and deny lists may specify particular users, the object's Class Manager, or the object itself. The lists can also include a special token that represents any LOID at all. The LOIDs of objects used to represent groups can also be contained in the lists. Group Objects simply represent a list of member LOIDs, providing methods for querying or modifying membership. Any user in the system can create their own group, listing whichever LOIDs they wish as members, and modifying membership dynamically over time. When a group object LOID is found on an access control list, all of the contained members are logically added to the list. For performance, the results of the membership lookup are cached, but with a short timeout (five minutes by default) so that group membership changes will be reflected relatively quickly. Groups provide one means for centralizing access control policy.

When a method call is received, the credentials it carries are checked by MayI and compared against the access control lists. For example, in the case of a delegated credential, the caller must have included proof of his identity in the call so that MayI can confirm that the credential applies. Multiple credentials can be carried in a call; checking continues until one provides access. Note that credentials provide an alternative way to define groups. If the group owner alone is on the access control list for a method, then he can give delegated credentials to all the members of the group, allowing them to call the method as well.

The default library MayI is configured when an object starts up. The configuration information is passed to it by its Class Manager, which in turn may have inherited the information, or part of it, from the user. For example, the user may have a default access control list for object-mandatory methods that all objects created on his behalf will inherit, while the Class Manager for those objects may specify additional access control lists specific to the particular kinds of objects they manage.

The form of access control provided by the default MayI is sufficient for some kinds of objects, such as file objects, but not for others. For example, Class Managers support a "deactivate" method that allows the caller to bring down an object managed by that Class Manager. Multiple clients of a single Class Manager Object may all need to call this method, but each should be allowed to deactivate only the objects he created. The default MayI doesn't have this ownership information. To solve this particular case, an additional MayI event handler is added to the Class Manager implementation that can check the arguments of the deactivate call against an internally maintained table of who created which object instance. The LRTL configurable, event-based protocol stack makes it easy to replace or supplement the default MayI with extra functionality such as this. The default MayI itself is relatively simple to modify if, for example, new forms of credentials or different

kinds of access control lists must be supported. With the Legion security architecture, these types of changes can be made on a local basis without affecting other parts of a Legion system.

### 3.3 Communication

A method call from one Legion object to another can consist of multiple Legion messages. Because Legion supports dataflow-based method invocation (as described in Section 2.1), the various arguments of a method call may flow into the target as messages from several different objects. The messages themselves are packetized and transmitted using one of a number of underlying transport layers, including UDP/IP, TCP/IP, or platform-specific message passing services (e.g., user-level message passing over an IBM SP2 switch).

It is at the level of Legion messages that we provide encryption and integrity services. When a Legion message is prepared for sending, various event handlers internal to the object are triggered in succession, as described in Section 2.3. One of these handlers implements a default message security layer. This layer inspects the implicit parameters accompanying a message to determine which security functions to apply.

#### 3.3.1 Message Security Modes

A message may be sent with no security, in *private mode*, or in *protected mode*. In both private and protected modes, certain key elements of a message (e.g., any contained credentials) are encrypted. The functional difference between the two modes is in how the rest of the message (body plus other implicit parameters) is treated. In private mode it is encrypted, whereas in protected mode only a digest is generated to provide an integrity guarantee. Unless private mode is already on, protected mode is selected automatically if a message contains credentials. This is a failsafe measure to prevent credentials from being transmitted in the clear.

The purpose of encrypting credentials is to protect bearer credentials from theft. Delegated credentials do not need to be protected, but the security layer does not examine the credentials at this level of detail. Moreover, the distinction between the two types of credentials can be misleading: If a delegated credential grants rights to a large group because further specificity is impossible, it may be desirable to protect it just like a bearer credential.

In addition to protecting credentials, both protected mode and private mode encrypt a *computation tag* contained in every Legion message, a random number token that is generated for each method call. All the messages that make up a given method call contain the same computation tag. The tag is used to assemble

incoming messages from multiple objects into a single method call and to identify the return value for a call made earlier. If an attacker knows the computation tag for a method call, he can forge complete messages containing arguments or return values, even without holding any credentials. The computation tag is treated as a shared secret, and is never transmitted in the clear unless “no security” mode is selected.

Private mode works by RSA-encrypting the entire message using the recipient’s public key. For efficiency, the RSA toolkit (RSAREF 2.0) only encrypts a random DES session key using RSA encryption; this key is then used to encrypt everything else in cipher-block-chaining (CBC) mode. The use of DES in CBC mode ensures that the message cannot be broken into pieces and recombined with other cryptotext to create a new valid message. For further efficiency, the sender and receiver cache the DES key so that if another message is sent in the same direction within a limited period, the DES key can be reused. The slow RSA encryption and decryption of the DES key can then be skipped.

Protected mode functions differently. First, a digest for the entire message is generated. Then, just the credentials and computation tag in the message are encrypted for the recipient. On receipt, they are decrypted, and the message digest is recalculated and compared with the transmitted value. An attacker cannot steal the encrypted block and use it to create another valid message because he is unable to create a digest that includes both his plaintext and the plaintext of the encrypted block. The latter can only be extracted by the intended recipient.

Protected mode is faster than private mode on large messages because digesting runs faster than DES encryption. Both modes pay the cost of RSA-encrypting a DES key, however.

Because the mode in use is stored in implicit parameters, it propagates through call chains. For example, a user can select private mode when calling an object. The calls that the object makes on behalf of the user will also use private mode, and so on down the line. In some cases this propagation is not desired, such as when a class object requests object creation on a Host Object. The class object always uses private mode for this call so that the new object’s private key is not exposed. However, the implicit parameters passed in this call will become the new object’s implicit parameters, and it may not need to run in private mode. The security layer recognizes a special nonpropagating implicit parameter to allow the specification of security for just a single message.

The security layer does not provide mutual authentication. The sender can be assured of the identity of the recipient, because only the desired recipient can read the encrypted parts of the message. The recipient usually doesn’t care who the actual sender is; its decisions are based solely on the credentials that arrived in the message.

### 3.3.2 Replay

Although credentials and computation tags cannot be extracted from a message by an eavesdropper, he can still attempt to replay a message. To prevent replay, each message includes a large random number and a timestamp. A recipient only accepts messages whose timestamps fall within a window based on the recipient's current time, e.g., thirty minutes into the past and ten minutes into the future. Old messages may be replays, excessively delayed, or victims of skewed clocks; messages from too far in the future also indicate overly skewed clocks. In these cases the recipient sends an exception back to the sender.

Assuming the message is arriving for the first time, the recipient then calculates how long it must log the random message number based on the message's timestamp and its own time-checking window. If another message then arrives with the same random number during the log period, it is rejected. In essence, the message timestamps allow coarse-grained detection of replays, while the random numbers provide the fine grain.

## 3.4 Object Management

So far we have discussed security at the level of Legion objects and facilities. Fundamentally, though, Legion software runs on existing operating systems with their own security policies. It is therefore also critical that the implementation of the Legion object model ensure that extra-Legion mechanisms cannot be used to subvert higher-level security mechanisms. Similarly, it is important to ensure that Legion does not break local security policies at a site. These issues are fundamental aspects of the Legion object management implementation—the interface between Legion core objects that represent resources, and the local system interfaces that provide access to resources.

Legion encapsulates the management of computational resources and data storage with Host and Vault Objects, respectively. The Host Object receives requests to create objects and controls them with the authentication and MayI mechanisms discussed earlier. It spawns new processes to run objects, monitors resource usage, and enforces allocation limits by killing user objects if necessary. In a complementary role, the Vault allocates OPRs as directories on the local file system or on other storage media. Authenticated objects use these allocated directories to store their persistent state. The Vault can reclaim managed storage space as necessary.

The local system administrator is generally concerned with who can create processes on his system via Legion, what those processes can do, and who pays for their resource use. If there is a security problem, he needs a way to trace the responsible party. On Legion's side, there is a need to prevent user objects from

interfering with one another or with system objects (e.g., Host and Vault Objects), and to maintain the privacy of persistent state (OPRs). The latter is particularly significant because objects store their private keys in their OPRs.

The needs of Legion are common to any multi-user operating system, and our approach to providing them is to leverage off of existing operating system services. In the following sections we show how we meet these needs and also satisfy two types of local system requirements.

### 3.4.1 Process Control Daemon

Our general strategy for isolating Legion objects from one another is to run them in separate accounts on the host system. The accounts that can be used for this purpose fall into two categories. For those Legion users who happen to have accounts on the local system, objects can run on their normal user accounts. For other users, there can be a pool of generic accounts that are assigned for Legion use. The generic accounts usually have minimal permissions (e.g., no home directory, no group memberships, etc.). The local Host and Vault Objects also have their own accounts.

Object creation requests arrive at the Host Object as normal method invocations, and can thus be controlled using the standard Legion access control mechanism for methods. For each request, the Host checks the credentials against the user LOIDs and groups that are allowed to create objects on it. If everything is acceptable, it next selects an account for the new object to run in; depending on the credentials in the creation request and its local configuration, it may choose a local user account or one of the generic accounts. The accounts are subject to scheduling and resource control just like CPU time, memory usage, and so on; an object's lease on an account, especially a generic account, is usually limited.

Before starting an actual process for the new object in the allocated account, the Host needs to change the ownership of the object's directory from the Vault user-id to the newly allocated user-id. The location of the directory that will contain the new object's persistent state is passed to the Host as part of the activation request (this location was obtained through a method on the local Vault performed by the object's creator, likely its class). Ownership of this directory must be changed to both protect the object's state from access by other objects (which will run under different user-ids), and to make the state accessible to the new object.

Finally, the Host needs to spawn the actual process that will execute the object on the appropriate account. To carry out this step, and to change ownership of the object's persistent state, the Host requires access to some privileged operations. However, the Host does not execute with root permissions. Access to these required privileged operations is encapsulated in a *Process Control Daemon* (PCD) that

executes on the host, providing services to the Host Object in a controlled fashion. The PCD is a small, easily vetted program that runs with root permissions. It is configured only to allow access by the host account. Two of its key functions are to permit changing directory ownership and to create new processes on a designated account. The PCD limits the accounts for which this can be done to a set configured by the local system administrator. The set includes the generic Legion accounts and potentially the accounts of local Legion users.

As the PCD starts the object running, the Host logs an audit trail using the X.509 information for the user whose credentials accompanied the request. The audit trail provides essential information if the new object misuses local resources.

While the object is allocated its account, the Host Object can reactivate it as necessary via the PCD (idle objects may deactivate themselves, or their class object may deactivate them). If the object has exceeded its use of local resources, the Host can request that the PCD kill it directly. When an object loses or relinquishes its use of an account, the Host object uses the PCD to change the ownership of its persistent state back to the Vault Object. If the object is reactivated later on a different account, ownership of the state can be changed to the appropriate user-id. After an account is reclaimed, the PCD terminates all processes running on it and generally cleans it up.

The Vault's storage is of course a limited resource as well, and scheduling and account policies apply to it. A Vault can clean up an object's state in several ways. Normally, the object's class will inform the Vault that the storage may be reclaimed (e.g., after a class migrates an instance off of a Vault, or deletes an instance, it calls the Vault's "delete storage" method). If the Vault initiates the clean-up, it proceeds conservatively, checking with the class object if possible and perhaps archiving the state before deleting it.

The implementation just described is sufficient for some local system administrators. Legion authentication is used to determine who gains access to local resources, and the resources made available are also constrained to those usable from a limited set of accounts. Detailed logging provides accountability.

### **3.4.2 Non-Legion Authentication**

One aspect of the previous approach which may be unacceptable at some sites is the use of Legion authentication mechanisms to control access to a host. For example, a site may require that only users with local accounts may access the system, and that those users must be authenticated by a locally adopted authentication system such as Kerberos [10]. Once authentication succeeds, though, normal Legion objects can be created. To make the discussion of how these requirements can be met concrete, we will use Kerberos as a sample authentication mechanism.

The Kerberos authentication protocol is fundamentally based on clients obtaining tickets for the use of services. Tickets are unforgeable tokens obtained from a distribution server through a protocol that involves the actual authentication of the user through password entry. To avoid requiring the repeated entry of passwords, a special Ticket Granting Ticket (TGT) is obtained by clients. This TGT is a credential that can then be used for a limited time to obtain further tickets that are required to access individual services. Clients can obtain specially marked TGTs that can be forwarded to proxies for use within a limited time period [9].

The use of these forwarded TGTs is the basis for employing Kerberos as an authentication mechanism within Legion. The TGT is sent in the implicit parameters of an object creation request to the eventual Host Object. The TGT is equivalent to a bearer credential, and it is treated as such by not being sent in the clear, etc.

The Host Object uses the TGT it receives to request a new TGT from the local Kerberos server. If authentication fails, it will not get a TGT. The new TGT grants access to a Kerberized version of the PCD. The Host proceeds to make an object creation request to the Kerberized PCD, supplying the local TGT and the name of the user's local account along with the standard job request information (e.g., the executable path, the path of the object's persistent state directory, etc.). The PCD performs the normal actions of changing the ownership of the directory for the object's state and spawning the object.

The TGTs expire relatively rapidly, and the Host discards them immediately after use. However, to support the continued management of local objects in the absence of the object owners, the PCD will perform certain limited actions on behalf the Host without a TGT. It can stop (i.e., kill) a managed object, it can restart the object, and it can switch the ownership of the object's state directory back and forth between the Vault ownership and user account ownership. The PCD ensures that only accounts of users authenticated via Kerberos are used.

An important restriction covers object restarts. The PCD will only restart the same objects under the same accounts that it did when TGTs were presented. It will not restart an object that is already running (thereby creating multiple copies), and it keeps track of its children to prevent this. These conditions prohibit the Host, or anybody contacting the Host, from leveraging off a previously authenticated use of the PCD. Consider the example of a user object running on a Host. If this object becomes deactivated, and then is needed to service a method called by a user other than its owner, it will need to be reactivated without a TGT. The PCD will reactivate the given object only if it had been previously started on the Host using a valid TGT. The PCD will not start additional objects of the same class (i.e., additional processes running the same executable), nor will it start the given object under a different user-id than was originally selected. The effect is the same as if the object had never been deactivated, but had instead continued to execute. This retains

the level of access control required by site administrators: processes can only be effectively started by authorized users, and then only through the locally mandated authentication mechanism. For the purposes of long-lived objects, we simply extend this to support the temporary suspension of objects created by authenticated users, and the subsequent reactivation of these processes by other clients.

The services provided by Kerberos (e.g., obtaining forwardable user credentials) are available in other systems, such as the Secure Sockets Layer (SSL). In fact, both Kerberos and SSL can be called through a generic interface: the Generic Secure Service Application Program Interface (GSSAPI) [8]. By using GSSAPI, straightforward extensions to other systems such as SSL are possible.

## **4 Policy Examples**

The Legion architecture presented in Section 2 is highly flexible, allowing the implementation of a wide variety of security mechanisms, important examples of which were described in Section 3. However, application developers and site administrators typically have higher-level policy specifications in mind when using software. The particular underlying mechanisms are less important, as long as the user can be assured that high-level policy requirements are being met. In this section, we consider illustrative examples of how the Legion system architecture and existing Legion tools can be organized to meet sample site and application policies.

### **4.1 Site-Local Policy Examples**

#### **4.1.1 Site Isolation**

As described in Section 2.2, Legion systems can consist of multiple domains, each possibly in a different organization or trust domain. For example, consider the example of a Regional Health Organization Network. Such a system would benefit from Legion's ability to allow enhanced collaboration, information sharing, and so on. However, this system would certainly interconnect institutions in disjoint trust domains. It is often desirable to system administrators contributing resources to a larger metasystem to ensure that certain site-isolation properties are guaranteed. For example, consider a site that makes resources available to Legion. It is managed by a local Legion administrator, who we will call Admin. A perfectly reasonable policy that would likely be required by Admin would be as follows: no matter how subverted any external sites in the Legion system might be, no intruder can invoke methods on local Legion resources as Admin. Such a policy is clearly desirable, since Admin is likely to have administrative control over critical local resources: who can use which machine, and for how long; who can access which

locally stored OPRs; etc. The ability to invoke methods as Admin is tantamount to complete control of the local Legion software.

The desired isolation policy can be achieved through a number of straightforward safeguards enabled by the Legion design. First and foremost, the local Legion resources should be started as a separate Legion domain (as described in Section 2.2). All of the core objects managing the local site can be started and configured by Admin, resulting in no external trust dependencies on outside systems. Clearly, to achieve the desired functionality of a metacomputer, the local domain will then need to be connected to some set of external Legion domains.

After this link to the external (and untrusted) system is made, Admin must take further precautions to prevent subversion of the local site. While invoking methods with bearer credentials based on Admin's proxy credential, Admin must be sure that no messages are sent to off-site objects. If we assume that any external site may be arbitrarily subverted and malicious, we cannot risk passing Admin's credentials outside the local site—they might immediately be used to break the isolation policy. However, simply stating that Admin should not pass credentials off-site is not generally good enough—Admin might make a simple mistake that could break the policy, so we would like automated enforcement of this safety measure. This automated enforcement is simple to achieve in Legion: Admin simply uses a version of the LRTL with the flexible protocol stack configured with an extra event handler for the message-send event. If a message is inadvertently directed off-site that contains Admin's credentials, the message is blocked and the event handler raises an exception. With this simple modification to Admin's Legion environment, he can be assured that his credentials will not be dispersed to untrustworthy off-site objects.

Ensuring that Admin does not communicate with off-site objects has a desirable secondary effect. Since Admin cannot communicate with external, untrustworthy sites, he cannot place critical objects such as his Refresh Object on resources at these sites (see Section 3.1.3). This benefit extends to an array of potentially critical, but not necessarily obvious, resources. For example, suppose Admin maintains a local Group Object listing the set of users that are allowed to start objects on local resources. If this object were allowed to execute on an untrustworthy site, its contents could be modified by a malicious resource owner, and local resource usage policy could be broken.

The two mechanisms described above, in combination with carefully configured access control for local core objects such as Hosts, Vaults, and critical Class Managers, ensure that the desired isolation policy will be met. Off-site objects will neither be able to generate nor steal Admin's credentials. External callers will be prevented from invoking unauthorized methods on local critical resources, ensuring that local access control is not tampered with, local resource usage policies are

not modified, and that security failures in other domains do not have dire consequences for the local site.

#### 4.1.2 Site-Wide Required Access Control

The Legion access control model as presented in Section 3.2 is based on the assumption that users will configure access control for their own objects. This concept adds a powerful level of flexibility to the system—for example, it makes arbitrary site-local resource access policies possible. However, on first examination it appears to relinquish the ability for a system administrator to set access control policies uniformly across his site. For example, the default Legion access control configuration does not grant the administrator user for a Legion domain access to other users' objects within the domain—there is no root user who can read any file or use any program in the domain. The inability to configure required site-wide access control policies may be unacceptable at some sites. However, the flexibility of the Legion architecture allows us to address this issue in a straightforward fashion, using the existing tools provided with the Legion software.

As an example of a site-wide required access control policy, we consider the problem of strictly limiting access to files by outside users. The Legion system defines a basic File Object that can be used to represent a file in the system. Access control for the normal Legion File Object is based on the default Legion ACL MayI mechanism, which places no restrictions on what LOIDs (i.e., what users) may be placed on access control lists. However, consider a site that wishes to enforce the policy that files may not be accessed by outside users. Effectively, we want a way to control which LOIDs may be placed on the ACL for local file objects. We can achieve this policy using the power of local Host Objects to control access to local resources. The Host Objects at the site (which are owned and controlled by the local administrator) are a point of resource access policy—they define which types of objects may run at the site. Using this feature, the site administrator can strictly limit the classes of objects that may run at the site. In particular, the allowable set of classes can be limited to those that are approved by the system administrator. The list of allowable classes can be configured to only include file objects with an alternate MayI layer—an extended version of the default ACL mechanism that also verifies that allowed LOIDs are in a well-known group containing only the local site users. Given this simple configuration, the site administrator can ensure that files are not inadvertently exported to outside users through Legion.<sup>2</sup> Furthermore, this approach generalizes to other site-wide access control restrictions, and other similar site-wide policy enforcement problems.

---

<sup>2</sup>Of course, the described approach does not prevent malicious users from exporting data off site in any number of ways, through Legion and otherwise.

### 4.1.3 Firewalls

Firewalls are a simple fact of life at many security-conscious institutions. While firewalls are not addressed explicitly in the Legion model, the Legion architecture is flexible enough to accommodate firewalls with ease. As is typical in firewall situations, a proxy on the firewall host is the natural solution. However, the ability to use custom versions of the Legion core objects, and the flexible protocol stack model of the LRLT, allow proxy-based solutions to be employed in Legion in an especially straightforward, user-transparent way.

Objects started on hosts behind a firewall automatically have a Proxy Object on the firewall host assigned to them by their Host Object (in some cases, each user might desire their own proxy object; in other cases, a shared proxy object is acceptable; either model is simple to support). The object address for a newly activated object behind the firewall that is reported to the object's Class Manager is actually the address for the Proxy Object—when callers of the object execute the binding process, they will be given the address of the Proxy Object. The Proxy Object then acts as a simple reflector, forwarding any received messages to their intended destinations behind the firewall. Use of the Proxy Object to forward outbound messages from callers behind the firewall is automated by a transparent add-in event handler in the LRTL protocol stack.

## 4.2 Application Policy Examples

### 4.2.1 Resource Selection Policy

In principle, a user of a metacomputer shouldn't need to care which resources are used to execute his jobs. In practice, however, the trustworthiness of the resources that are selected for certain applications is of critical interest to the user. Policies regarding which resources may be used to execute objects are logically localized within the Class Managers of a user's object classes. Any site selection policy can be encoded in a user's Class Manager Objects, giving the user total control over the selection and use of trustworthy sites.

Although this problem is solved cleanly at the architectural level in Legion, we deemed this issue of site selection for application users important enough to warrant special features in the default Class Manager Object reference implementations. All default Class Managers in the Legion implementation check for certain implicit parameters that can be used to limit resource selection. By setting these implicit parameters in his Legion environment (using a provided tool), the user can configure a resource selection policy that will propagate to all "create instance" methods called on Class Manager objects on behalf of the user. Of course, the architectural principle that users can encode any resource selection policy they

wish in their own Class Manager implementations still holds; in fact, a convenient model for such customization is supported by the default Class Manager's ability to be configured to use an external *Scheduler Object* with a well-known interface. However, in the common case, where a user can generate a list of sites that he deems trustworthy and indicate this in his environment, the default implementation provides the needed mechanism to implement a basic, effective resource selection policy.

#### 4.2.2 Customized Access Control Policies

The default, ACL-based access control mechanism provided in the LRTL basic MayI implementation is useful for specifying many common access control policies. For example, the basic Legion File Object has methods such as *read*, *write*, and *truncate*. By specifying allow and deny lists of users and groups for these methods, we can achieve the traditional file access policies familiar to users of common existing file systems. This is also true for other Legion services, such as Context Space. However, in some cases, access control lists are not sufficient to specify the required policy. Consider the example of an object that represents a database of patient records in a hospital. Suppose that this database object has methods to create, query and update the record for a patient. We would certainly want all of the doctors in the hospital to have access to these methods. However, we might also want to enforce the policy that a patient's record is only available to his health care providers. Access control lists do not let us express this policy.

Solving this problem is straightforward in Legion. Since MayI is an event in the configurable protocol stack, we can introduce a new MayI event handler to enforce the desired policy. The new MayI handler would check the record in question on query and update methods against the method caller (indicated as the signer of a credential granting access to the method in question). If the caller is listed as one of the doctors for the patient whose record was being accessed, the call would be allowed to proceed. Otherwise, MayI would reject the call and raise an exception.

In practice, we employ exactly this sort of add-on MayI functionality in the reference implementations of the Legion core objects. For example, Vault objects provide access to object persistent state. All users of a given Vault need to access some of the OPRs contained in the Vault, but we want to ensure that users can't access one another's OPRs. We use an extra MayI layer to ensure that only the appropriate object owners (or the owner of the Vault itself) can access OPRs. Similar functionality is used to control object management operations on Host Objects and Class Managers.

## 5 Related Work

Two projects that incorporate security into large-scale distributed computing platforms are Globus and WebOS. Globus [2] is a “bag of services” model for meta-computing, in contrast to Legion’s integrated environment approach. Whereas Legion security is fundamentally built into the architecture of the system, Globus security services are provided as add-on modules. Other Globus toolkit modules vary in the degree to which they integrate with, or use the services of, the security modules. In Legion, we have adopted the approach of defining a set of simple but powerful abstractions that may be easily composed to implement new security policies, as our examples demonstrate. This approach is inherently more flexible and adaptable.

CRISIS [1] is the security architecture for WebOS. WebOS is fundamentally different from Legion in terms of the basic services provided. WebOS provides a single, traditional file system and a fixed interface for authenticated remote process creation. CRISIS defines careful, effective security policies for these basic services. However, the CRISIS solution does not provide a means for easily developing security policies for new mechanisms as they are added to WebOS, nor does it provide a means for modifying the security policies supported for the existing services.

Two other projects related to security efforts in Legion, although not with the focus on metasecurity, are Java and CORBA. The computational model of Java [3] (JDK 1.2) requires identity and authentication in order to execute digitally signed code downloaded from a remote site. The JDK provides per-class (or per-application) protection domains. However, it differs significantly from Legion in its lack of support for per-site security mechanisms, delegation, and user authentication.

The security model of CORBA [11] encompasses identification and authentication, authorization and access control, auditing, security of communication, non-repudiation, and security information administration. Typically, an ORB vendor implements CORBA security using existing technology such as GSSAPI, Kerberos, and SESAME. Many of the goals of the CORBA security model are similar to the goals of the Legion security model, including simplicity, scalability, usability, and flexibility. However, CORBA is not a metacomputing system—it does not construct an operating system-like environment using underlying distributed resources. Given this fundamental difference in target use, CORBA does not address the metacomputing security problem.

## 6 Conclusions

We have presented the basic security architecture of the Legion system, and we have demonstrated that our design is sufficiently flexible to accommodate a wide variety of security-related mechanisms. This flexibility is critical to the successful deployment and use of metacomputing software. One-size-fits-all software dictated by a single group will never satisfy the requirements of the wide range of users and resource providers in a large-scale, cross-domain environment. We have also demonstrated that flexibility does not come at the price of complete lack of control. Within the flexible Legion framework, we showed how a number of important site-wide and application-wide security policies could be achieved. Naturally, the set of policies presented is only a small fraction of the policies that will be needed across the complete Legion environment.

The Legion system, including the security features described here, is currently publicly available. It is widely deployed on hundreds of machines at dozens of sites spanning multiple trust domains. Key portions of the software, such as the PCD described in Section 2.2, have been vetted and approved by system administrators at sites such as the San Diego Supercomputing Center and the US Naval Oceanographic Office (NAVO). In the future, we plan to continue deployment of Legion, developing additional mechanism and adapting to new site-local policies as required. We are also in the process of measuring the performance impact of key Legion security mechanisms.

## References

- [1] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. CRISIS: A wide area security architecture. In *Seventh USENIX Security Symposium*, January 1998.
- [2] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *Fifth ACM Conference on Computers and Communications Security*, November 1998.
- [3] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, December 1997.
- [4] Andrew Grimshaw, Michael Lewis, Adam Ferrari, and John Karpovich. Architectural support for extensibility and autonomy in wide-area distributed object systems. Technical Report CS-98-12, Department of Computer Science, University of Virginia, Charlottesville, Virginia, June 1998.

- [5] Andrew S. Grimshaw and William A. Wulf. Legion: A view from 50,000 feet. In *Fifth IEEE Symposium on High Performance Distributed Computing*, August 1996.
- [6] Andrew S. Grimshaw and William A. Wulf. The Legion vision of a world-wide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.
- [7] Mike Lewis and Andrew Grimshaw. The core Legion object model. In *Fifth IEEE Symposium on High Performance Distributed Computing*, August 1996.
- [8] J. Linn. Generic security service application program interface, version 2. RFC 2078, January 1997.
- [9] B. Clifford Neuman. Proxy-based authorization and accounting for distributed systems. In *Thirteenth International Conference on Distributed Computing Systems*, pages 283–291, May 1993.
- [10] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.
- [11] Object Management Group. CORBA services: Common object services specification, security service specification. Version 97-12-12, 1998.
- [12] C.L. Viles, M.J. Lewis, A.J. Ferrari, A. Nguyen-Tuong, and A.S. Grimshaw. Enabling flexibility in the legion run-time library. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 265–274, June 1997.
- [13] William A. Wulf, Chenxi Wang, and Darrell Kienzle. A new model of security for distributed systems. Technical Report CS-95-34, Department of Computer Science, University of Virginia, Charlottesville, Virginia, August 1995.