A New Model of Security for Distributed Systems

Wm A. Wulf
Chenxi Wang
Darrell Kienzle

Abstract

With the rapid growth of the information age, open distributed systems have become increasingly popular. The need for protection and security in a distributed environment has never been greater. The conventional approach to security has been to enforce a system-wide policy, but this approach will not work for large distributed systems where entirely new security issues and concerns are emerging. We argue that a new model is needed that shifts the emphasis from "system as enforcer" to user-definable policies. Users ought to be able to select the level of security they need and pay only the necessary overhead. Moreover, ultimately, they must be responsible for their own security.

This research is being carried out in the context of the Legion project. We start by describing the objectives and philosophy of the overall project and then present our conceptual model and design decisions. A set of technical challenges and related issues are also addressed.

1 Introduction

High speed networking has significantly changed the nature of computing, and specifically gives rise to a new set of security concerns and issues. The conventional security approach has been for "the system" to mediate all interactions between users and resources, and to enforce a single system-wide policy. This approach has served us well in the environment of a centralized system because the operating system implements all the key components and knows who is responsible for each process.

However, in a large distributed system several things have changed:

- Distributed Kernel: There is no clear notion of a single protected kernel. The path between any two objects may involve several machines that are not equally trusted.
- System Scope and Size: The system is usually much larger than a centralized one. It may very well be a federation of distinct administrative domains with separate authorities.
- Heterogeneity: The system may involve many subdomains with distinct security policies, channels that are secured in several ways, and platforms with different operation systems.

The intricate nature of distributed system has fundamentally changed the requirement of system security. We are investigating a new model of computer security — a model appropriate to large distributed systems in the context of Legion — a system described below.

Users of Legion-like systems must feel confident that the privacy and integrity of their data will not be compromised — either by granting others access to their system, or by running their own programs on an unknown remote computer. Creating that confidence is an especially challenging problem for a number of reasons; for example:

- We envision Legion as a *very* large distributed system; at least for purposes of design, it is useful to think of it as running on millions of processors distributed throughout the galaxy.
- Legion will run *on top of* a variety of host operating systems; it will not have control of the hardware or operating system on which it runs.
- There won't be a single organization or person that "owns" all of the systems involved. Thus no one can be trusted to enforce security standards on them; indeed, some individual owners might be malicious.

No single security policy will satisfy all users of a huge system — the CIA, NationsBank, and the University of Virginia Hospital will have different views of what is necessary and appropriate. We cannot even presume a single "login" mechanism — some situations will demand a far more rigorous one than others. Moreover we cannot anticipate all the policies or login mechanisms that will emerge; both will be added dynamically. And, for both logical and performance reasons, the potential size and scope of Legion suggests that we should not have distinguished "trusted" components that could become points of failure/penetration or bottlenecks.

Running "on top of" host operating systems has many implications, but in particular it means that in addition to the usual assumption of insecure communication, we must assume that copies of the Legion system itself will be corrupted (rogue Legionnaires), that some other agent may try to impersonate Legion, and that a person with "root" privileges to a component system can modify the bits arbitrarily.

The assumption of "no owner" and wide distribution exacerbates these issues, of course. Since Legion cannot replace existing host operating systems, the idea of securing them all is not a feasible option. We have to presume that at least some of the hosts in the system will be compromised, and may even be malicious.

These problems pose new challenges for computer security. They are sufficiently different from the prior problems faced by single-host systems that some of the assumptions that have pervaded work on computer security must be re-examined. Consider just two such assumptions. The first is that security is absolute; a system is either secure or it is not. A second is that "the system" is the enforcer of security.

In the physical world, security is never absolute. Some safes are better than others, but none is expected to withstand an arbitrary attack. In fact, safes are rated by the time they resist particular attacks. If a particular safe isn't good enough, its owner has the responsibility to get a better one, hire a guard, string an electric fence, or whatever. It isn't "the system", whatever that may be, that provides added security.

Note that we said that users must feel "confident"; we did not say that they had to be "guaranteed" of anything. Security needs to be "good enough" for a particular circumstance. Of course, what's good enough in one case may not be in another — so we need a mechanism that first lets the user know how much confidence they are justified in having, and second provides an avenue for gaining more when required.

The phrase "the trusted computing base" (TCB) is common when referring to systems that enforce a security policy. The mental image is that "the system" mediates all interactions between users and resources, and for each interaction decides to permit or prohibit it based on consulting a "trusted data base"; the Lampson access matrix [] is the archetype of such models. Even communications, which is inherently insecure, is usually presumed to be inside the perimeter and the system is considered to be responsible for implementing secure communication on top of the insecure base.

As with the previous assumption, this one just doesn't work in a Legion-like context. In the first place there isn't a single policy, new ones may emerge all the time, and the complexities of overlapping/intersecting security domains blur the very notion of a perimeter to be protected. In the second place, since we have to presume that the code might be reverse-engineered and modified, we cannot rely on the system enforcing security — or very much of anything, for that matter.

Moreover, security has a cost in time, convenience, or both. The intuitive determination of how much confidence is "good enough" is moderated by cost considerations. As has been observed many times, one reason that extant computer systems have not paid more attention to security is that the cost, especially in convenience, is too high. These prior systems took the "security is absolute" approach, and everyone paid the cost regardless of their individual needs. To succeed, our model must scale — it must have essentially zero cost if no security is needed, and the cost must increase in proportion to the extra confidence one gains.

The above observation calls for rethinking some very basic, often stated assumptions — that is, a change in the way of thinking and a shift in security paradigm. In the rest of the paper, we suggest a new security model that differs from the traditional approach. We also illustrate ideas to deal with the issues raised above, as well as others. Before proceeding to describe our plan of attack, the following describes the Legion system to provide context.

2 Background – The Legion Project[1]

The Legion project at the University of Virginia is an attempt to provide system services that create the illusion of a single virtual machine, a machine that provides secure shared object and shared name spaces, high performance via both task and data parallelism, application adjustable fault-tolerance, improved response time, and greater throughput. Legion is targeted towards wide-area assemblies of workstations, supercomputers, and parallel supercomputers. Such a system, if constructed, will unleash the integrated potential of many diverse, powerful resources which may very well revolutionize how we work, how we play, and in general, how we interact with one another.

The potential benefits of Legion are enormous. We envision (1) more effective collaboration by putting coworkers in the same virtual workplace; (2) higher application performance due to parallel execution and exploitation of off-site resources; (3) improved access to data and computational resources; (4) improved researcher and user productivity resulting from more effective collaboration and better application performance; (5) increased resource utilization; and (6) a considerably simpler programming environment for the applications programmers. Indeed, it seems probable to us that the NII can reach its full potential only with a Legion-like infrastructure.

2.1 The Legion Object Model and System Philosophy

Legion is an object-oriented metasystem[2]. The principles of the object-oriented paradigm are the foundation for the construction of Legion; All components of interest in Legion are objects, and all objects, including classes, are instances of classes. Use of the object-oriented paradigm enables us to exploit the paradigm's encapsulation and inheritance properties, as well as benefits such as software reuse, fault containment, and reduction in complexity.

Hand-in-hand with the object-oriented paradigm is one of our driving philosophical themes: we cannot design a system that will satisfy every users' needs, therefore we must design an extensible system. This philosophy manifests itself throughout, particularly in our use of delayed binding and what we call "service sliders". Consider security. There is a trade-off between security and performance (due to the cost of authentication, encryption, etc.). Rather than providing a fixed level of security - with the result that no one will be happy, we allow users to choose their own trade-offs by implementing their own policies or using existing policies via inheritance. Similarly users can select the level of fault-tolerance that they want - and pay for only what they use. By allowing users to implement their own or inherit services from library classes we provide the user with flexibility while at the same time providing a menu of existing choices.

2.2 Design Objectives and Restrictions

We have the following design objectives, against which we measure our success; site autonomy; an extensible core; scalability; easy-to-use, seamless computational environment; high performance via parallelism; single, persistent

1. We cannot adequately present Legion in the few pages below. For more information see our web page at http://www.cs.virginia.edu/~legion/

2. This does not imply that Legion supports only object-oriented languages. Legion will support traditional languages such as C and Fortran as well.

namespace; security for both users and resource providers; manage and exploit resource heterogeneity, and fault tolerance.

In addition to the goals above, two constraints restrict our design — we cannot replace host operating systems, and we cannot legislate changes to the interconnection network.

To accomplish the goals, many technical, political, sociological, and economic issues need to be resolved. In this paper we attempt to address the security aspect of the Legion project.

3 The Security Model

In this section we describe a design for the security model in Legion. The model, following closely to the Legion philosophy, responds to the issues raised in the introduction. We first present the design guidelines and principles. We discuss the trade-offs and our design decisions. We then explain how the model works, in particular how it can be used to enforce discretionary policies.

The premise here is that we cannot, and indeed should not, provide a guarantee of security. What we can and should do is (1) be as precise as possible about the degree of confidence a user can have, (2) make that confidence "good enough" and "cheap enough" for an interestingly large selection of users, and (3) provide a context that allows the user to gain the additional confidence they require with a cost that is intuitively proportional to the added confidence they get.

3.1 Design Principles

The Legion Security model is based on three principles:

- first, as in the Hippocratic Oath, *do no harm*!
- second, *caveat emptor*, let the buyer beware.
- third, *small is beautiful*.
-

Legion's first responsibility is to minimize the possibility that it will provide an avenue via which an intruder can do mischief to a remote system. The remote system is, by the second principle, responsible for ensuring that it is running a valid copy of Legion — but subject to that, Legion should not permit its corruption.

The second principle means that in the final analysis users are responsible for their own security. Legion provides a model and mechanism that make it feasible, conceptually simple, and inexpensive in the default case, but in the end the user has the ultimate responsibility to determine what policy is to be enforced and how vigorous that enforcement will be. This, we think, also models the "real world"; the strongest door with the strongest lock is useless if the user leaves it open.

The third principle simply means, given that one cannot absolutely, unconditionally depend on Legion to enforce security, there is no reason to invest it with elaborate mechanisms. On the contrary, at least intuitively, the simpler the model and the less it does, the lower the probability that a corrupted version can do harm. The remainder of the paper describes such a simple, albeit evolving model. The description is discursive, but a much shorter, formal definition will be forthcoming.

As noted above, Legion is an object-oriented system. Thus,

- the unit of protection is the object, and
- the "rights" to the object allow invocation of its member functions (each member function is associated with a distinct right).

This is not a new idea; it dates to at least the Hydra system in the mid 1970's [6] and is also in some proposed CORBA models [10]. Note, however, that it subsumes more common notions such as protection at the level of file systems. In Legion, files are merely one type of user-defined object that happen to have methods read/write/seek/etc. Directories are just another type of object with methods such as lookup/enter/delete/etc. There is no reason why there must be only one type of file or one type of directory and, indeed, these need not be distinguished concepts defined by, or even known to Legion.

The basic concepts of the Legion Security Model are minimal; there are just four:

- every object provides certain known member functions (that may be defaulted to NIL); the ones we will describe here are "MayI," "Iam," and "Delegate.".
- there is a "responsible agent" (RA) associated with each operation. The RA is someone who can be held accountable for the particular operation. There are a certain set of member functions associated with an RA object. User-defined objects can play the role of RA by supplying these member functions.
- every invocation of a member function is performed in an environment consisting of a pair of (unique) object names — those of the operative responsible agent, and "calling agent", CA.
- there are a small set of rules for actions that Legion will take, primarily at member function invocation. These rules are defined informally here.

The general approach is that Legion will invoke the known member functions (MayI, etc.), thus giving objects the responsibility of defining and ensuring the policy. Precisely how this happens is detailed in the following sections.

3.2 Protecting Oneself — Privacy

In Legion users are responsible for their own security. They are the ones who decide how secure their applications ought to be, and from there, which policy is to be enforced and how rigorous the enforcement should be.

For example, a truly paranoid user's object can (and should, if they deem it important) include code in every method to authenticate the caller and to determine whether that caller has the right to make this call. This cautious user most likely will not be satisfied unless some elaborate authentication scheme is used to identify the caller.

For many users, however, this degree of caution is unnecessary and some delegation to the Legion mechanism is appropriate — for example, rather than engaging in an authentication dialog with the caller, an object might trust that the CA field of the environment is correct. In the following we'll describe how the model facilitates appropriate, situation-specific delegation; for readability we'll proceed in several steps, each of which adds a bit more detail and refinement.

Our first objective is to have policies defined by the objects themselves. At the same time, we don't want to have to include policy-enforcement code in every member function unless the object is particularly sensitive. So, instead, we require that every class define a special member function, "MayI" (this can be defaulted, but we'll ignore that for now). MayI defines the security policy for objects of that class. Conceptually at least, Legion will automatically call the MayI function before every member function invocation, and will permit that invocation only if MayI sanctions it (see figure 1).

We'll refine this in a moment to be both more efficient and more powerful — but note how this simple idea begins to meet our objectives. First, it permits the creator of an object class to define the privacy policy for objects of that class; there is no system-wide policy. Second, it is fully extensible — when a user defines a new class its member functions become the "rights" for that class and its MayI function/policy determines who may exercise those rights. Third, it is fully distributed; there is no distinguished trusted data base (each MayI may consult a database if it chooses, but there is no "distinguished" one(s)). Fourth, it is not particularly burdensome; users can default MayI to "always OK", inherit a MayI policy from a class they trust, or write a new policy if the situation warrants it. Fifth, the code for implementing the security policy is localized to the MayI function rather than distributed among the member functions. Finally, the default "always OK" policy can be optimized so that there is no overhead at all associated with the mechanism.

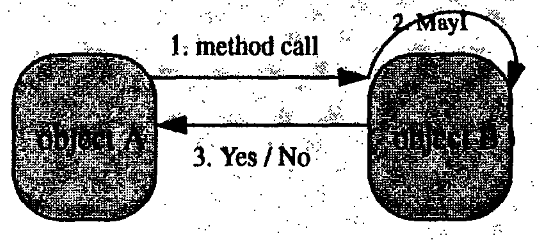

Figure 1

3.3 Authentication

The previous discussion left one question unanswered: who or what is the "I" that the MayI function grants access to? Indeed, the request must first be authenticated to identify the principal that uttered it, and then authorized only if the principal has the right to perform the operation on the object. The principal behind the request could be human users, software programs, or compound identities such as delegations, roles and groups.

Authentication in Legion is aided by the use of Legion environment. Recall that the environment contains two object identifiers, namely the calling agent (CA) and the responsible agent (RA). The CA is the object that initiated the current method call. The RA is a generalization of the "user id" in conventional systems; for the moment it is adequate to think of it as identifying the user or agent who was responsible for the sequence of method invocations that lead to the current one.

In the general spirit of our approach, the authentication of the caller and caller's context can be anything that the MayI function demands — and in sensitive cases, that is just as it should be. In most cases, however, "I" will be simply CA, or RA, or any subset of the two. Indeed, by analogy with familiar systems where "I" is the user, that subset may be just RA.

Legion makes a specified level of effort to assure the authenticity of the environment IDs; this effort should be adequate for most purposes. However, in the spirit of the second principle, we expect that MayI functions with extraordinary security concerns will code their own authentication protocols by, for example, calling back to the caller, and/or responsible agent. To make this possible, we require every Legion object to supply a special public member function — "Iam" for authentication purposes. In the same principle as "MayI", "Iam" could be optimized to NIL.

Legion bases authentication on public-key cryptography in the default case. Knowledge of the private key is the proof of authenticity. In addition, a set of general principle authentication protocols will be provided as the system standard. "Iam" can choose to support all or none of them. Other more elaborate protocols could be negotiated between objects and made known to the "Iam" function. Objects unprepared to adequately authenticate themselves are ipso facto not to be trusted. The result of "Iam" can be cached for future reference, but that is an implementation choice and is beyond the scope of this paper.

3.4 Login

The avenue via which Legion users authenticate themselves to Legion is the Login procedure. Login establishes user's identity as well as creating a responsible agent object for the user. The login procedure is therefore the building block for future authentication, delegation and creating of compound identities.

By the same design principle, Legion should not mandate a single "Login" mechanism. Typically, there is a login object that will be invoked when a user first logs in. This login object engages in a login dialog with the user and, if satisfied, declares itself to be the responsible agent. Actually, any Legion object may declare itself to be the current responsible agent should it choose. It simply does so by executing a "RA = me" command (environments are stacked, so that a return from an object executing this command will revert to the previous RA).

There are many advantages to why we shouldn't make this "login" mechanism universal. For example, logging on to Legion in UVa may require only a simple password while Legion in CIA might demand their users to submit fingerprints or retinal scan information. Users can define their own login class with varying degrees of rigor in the login dialog, specific to their needs. The "login" mechanism can also be easily inherited or defaulted to some simple scheme.

How do we know that a particular login class is to be trusted? We don't, in general. The MayI function of another class need not believe the login! After interrogating the class of the responsible agent the MayI function may reject the call if the login is either insufficiently rigorous, or simply unknown to this MayI. As in the infamous "real world", trust can only be *earned*.

3.5 Delegation

In all security models one must consider how rights propagate; Can a principal hand all or some of its authority to another, and how can a principal restrict its authority? For example, a user on a workstation may wish to delegate the "read" right on her files to the C compiler. The compiler can then access files on her behalf as long as the delegation still stands, much in the same way the user may wish to delegate. Just as the basic security policy is embedded in MayI and not in Legion, our model does not answer this question — but it does provide a uniform way for the user to answer it.

We require every Legion object to have another public method, "Delegate." The parameters to Delegate are the ID of the object to which rights should be delegated, and a set of restrictions that limit those rights. For example, a user object A wants to invoke a compiler C and pass the "read-only" right on file F to C. To accomplish this, A must invoke the "Delegate" function of F to request such a delegation. Using a C++ like notation, but prefixing it with the name of the executing object and a colon, this is:

```
    A: F.Delegate(C, read);
```

F, upon receiving the above request, can either grant the delegation, reject it, or grant delegation of a more restricted authority than what is requested. Granting delegation may result in storing some information locally or in creation of a new entry in some database (for example, an access control list) known to MayI.

A then instructs C to compile the file by passing it the ID of F.

```
    A: C.compile(F)
```

When C attempts to read F, F's MayI is invoked. MayI recognizes this delegated authority either by looking up some local information or consulting some external database. The operation is thus permitted. However, if C attempts to invoke any of F's other methods, F will disallow this.

Our philosophy is that delegation policy is a part of the discretionary policy which should be defined by the object itself. Indeed, delegation policies can be arbitrarily complex or light weight. Classes that want to take extreme precautions against delegation may choose not to support delegation at all — this is the default. Alternatively, users can write their own delegation functions or inherit appropriate ones from existing classes — for example, by including a time limit as part of the access database, delegation can be made to expire after certain time period.

So far we have discussed three security-related functions: MayI, Iam and Delegate. They are user-defined functions, together, quite elegantly, they form a guard or reference monitor upon which any discretionary policy can be defined. In addition,

- "MayI", "Iam" and "Delegate" can be defaulted to NIL and hence will impose no overhead. And indeed, many classes will favor the default case for performance reasons.
- when these functions are non-NIL, they enforce user-definable policies rather than some global Legion-defined one,
- these functions can be as simple or as elaborate as the user feels necessary to achieve their comfort level — the "service slider" approach again.

3.6 Licenses

MayI is a relatively costly operation that may involve consultation of external databases and extra message exchanges for authentication. Invoking MayI for each method invocation is both technically and conceptually inefficient; a slight modification both removes this inefficiency and expands the power of the model. Rather than returning a simple boolean, MayI is expected to return a record; this record is the key to invoke any member function other than MayI — and thus is related to what are called "capabilities" and "tickets" in other schemes. We call the record a 'license' because it grants access to the object under terms and conditions defined within it. The conceptual form of a license is:

```
RA, CA:                  UID;
rights:                  array of boolean;
t:                       life time;
n:                       integer;
f:                       function;
cr,cc,ct,cn,cf:          boolean;
```

The "rights" are bits that map one-to-one onto the object's member functions. To invoke a particular member function, its corresponding rights bit must be set — if it isn't, the attempted call is not permitted. The remainder of the fields define the conditions under which the license is valid; operationally this means:

if cr is true, the environment's responsible agent must be equal to the license's RA field.

if cc is true, the environment's calling agent must be equal to the license's CA field.

if ct is true, the current time must be less than t[1].

if cn is true, the number of previous member function invocations under this license must be less than n.

If any of these fail, MayI is reinvoked — that is, a failure does not necessarily preclude the method invocation, it just ensures that MayI is given another chance to assess the trust in "I". A whole spectrum of choices are given by the terms and limits specified in the license. For example, at one extreme, by setting cn = true and n = 1, MayI is invoked on every call. At the other extreme, if cr = cc = ct = cn = cf = false, and all the rights bits are set, everything is permitted to anyone (This is in fact a classical capability scheme). Between these extremes are a rich variety of options in terms of both who may use the license and how often MayI is reinvoked.

1. Both the time of the call and time t stored in the license are based on the clock of the machine hosting the object being invoked.

After these checks are made, if they do not fail, the cf bit is tested and the function 'f' may be invoked:

if cf is true, call f, which returns one of : OK, NO, or "invoke MayI"

The function f is a hook for users to define arbitrary constraints or checking mechanisms in addition to the ones specified in the license. For example, a typical f function can restrict access to any member function until, say, after 6 pm. In most cases, applications may want to default f to NIL while others can implement f to do anything they wish.

If f is invoked and the OK value is returned, the method invocation is permitted. If the returned value is NO, the attempted call fails just as though the appropriate rights bit were not set. In the third case the current license is revoked and MayI is called to create a new one.

This mechanism permits the user not only to define the security policy for an object, but also to make trade-off decisions about the cost of that security. As in the real world, cost is a legitimate concern, and we expect MayI may be a rather costly operation while checking the license will be inexpensive. It is easy to posit situations where security is of no concern at all — as in reading the system clock — and a "heavy" mechanism would be totally inappropriate. Conversely more sensitive objects may deserve frequent "lightweight" checks, infrequent more thorough checks, or a full reauthentication with retinal scan on every access. All of these are supported by the model.

3.7 Cache the licenses

We would like to briefly discuss what might be considered an implementation issue, but in fact has impact on the model's power. In one obvious implementation, licenses would be stored in the user's (caller's) memory; if precautions aren't taken the user could modify them in illegal ways — adding rights, increasing the time limit, etc. The usual way to prevent this is to encrypt the licenses so that any modification results in garbage, but we are thinking of another alternative. The alternative has a number of advantages, including enabling revocation ("taking back" rights that were previously granted) and better protection.

The idea is relatively simple. When MayI returns a license, L, Legion will cache the license in the called object's address space, indexed by the RA, and CA UIDs. Subsequent requests from the caller will result in a local cache lookup at the site of the called object. The method call and the rights presented in the license are matched to check the validity of the call. Note this means that the license is never in the caller's address space and hence cannot be modified by it (if the called object tramples on the license, well, caveat emptor!). This is not to say, the information in the license cannot be disclosed to the caller; since it cannot be used in any way, a copy of the entire license can be made available upon request.

Figure 2 illustrates an example in which object A calls B on behalf of user Alice. Previously, the user executed "RA = me" to set the RA field to Alice. Next, the function call from Alice to A has an identical RA and CA field; both are "Alice". When a local cache lookup is performed by A, a license is found and subsequently checked. A then calls B, and the RA field remains the same while the CA is now "A". The same cache checks are performed by B upon receipt of A's invocation request.

There are a number of advantages to this scheme:

- the caller can do no mischief to the license since it does not have write access to it
- revocation of licenses is trivial, and under control of the called object — just delete some (or all) of the licenses in its local cache
- the user need not be aware that any of this is happening — they can just invoke the appropriate member functions
- the default (no protection) case can be optimized — no caching or checking needs to be done

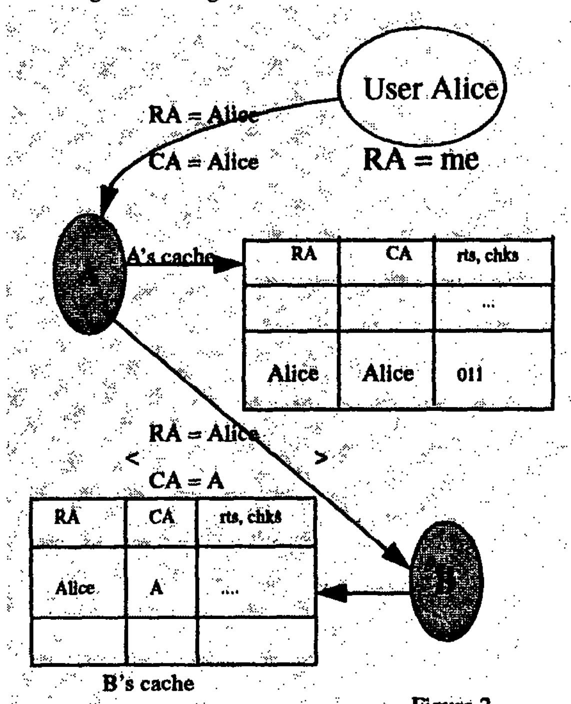

Figure 2

4 Mandatory Policies

Mandatory policies, such as multi-level security, presume that the parties involved may be conspirators and impose some sort of check by a third party — usually "the system" — between caller and called objects. Generally this imposition is completely dynamic — every call is checked.

In the Legion context, of course, we eschew the idea of a system-wide policy. Thus we need a safe mechanism that interposes an arbitrary enforcer of an arbitrary policy between caller and called object. Interestingly, when combined with inheritance, the MayI function already discussed provides half the answer, albeit in a somewhat different way.

Imagine that a new mandatory security regime is to be created. An obvious consideration is that the enforcer, which we'll call the "security agent" must know about all of the kinds of objects in its domain — it cannot enforce "no write down" if it doesn't know what a "write" to a specific object is, for example. Thus we'll begin with the presumption that a good security agent simply won't allow calls on objects of unknown pedigree.

Given that, it is reasonable to presume that the security agent can derive subclasses for the objects that it does know about; in these subclasses the security agent can inherit a MayI function of its choosing — and specifically one that invokes the security agent to verify the validity of each inward call. All the objects, and only the objects that are instances of these derived classes will be permitted in this security agent's regime.

As noted above, this solves half the problem — the security agent is invoked whenever an object under its control is called. We need to add the symmetric capability for outward calls; thus we add a method IWantTo that, if non-null, is invoked by Legion whenever an object attempts to make a call on another object. Now, by deriving a class that defines both the MayI and IWantTo methods, the security agent can be ensured that it gets invoked on every call involving one of the objects under its control.

Note that while the usual mechanism for enforcing mandatory policies is done completely at run time, the one we have described is partially a "compile time" (or "link time") mechanism — that is, the time at which the MayI and IWantTo methods are bound into the subclass. Although this seems almost required by the rejection of a single system-wide policy, it might raise concerns over the possibility of intentional corruption of the mechanism. This is a subtler topic than can be handled in detail here, but the reader may gain some comfort from the observation that we have inverted the usual (temporal) relation between defender and attacker. In the traditional scenario the defender of security puts out a system which the attackers then may analyze and attack at leisure. In this case, if the attack is to be mounted from within an object that the security agent has "wrapped" with its own MayI and IWantTo functions, the attacker must put their code out first without knowledge of how it will be wrapped. In this case, the security agent has the advantage of examining the purported attackers code before deciding whether to allow it into its security regime.

Finally, although we won't discuss it here, obviously we can define a license mechanism for IWantTo that is analogous to that for MayI, with the analogous benefit — IWantTo can get involved as much or as little as it deems appropriate.

5 Is There An Imposter In The House?

In a large distributed system such as we envision, it is impossible to prevent corruption of some computers. We must presume that someone will try to pose as a valid Legion system or object in order to gain access to, or tamper with other objects in an unauthorized way. That is why, in the final analysis, the most sensitive data should not be stored on a computer connected to any network, whether running Legion or not.

On the other hand, perhaps we can make the probability of such mischief sufficiently low and its cost sufficiently high to be acceptable for all but the most sensitive applications. We have formulated a number of principles that form a basis for our ongoing research. They are:

1. *Defense in depth*: There won't be a single silver bullet that "solves" the problem of rogue Legionnaires, so each of the following is intended as an independent mechanism. The chance that a rogue can defeat them all is at least lower than defeating any one separately.

2. *Least Privilege*: Legion will run with the least privilege possible on each host operating system. There are two points to this: first, it will reduce the probability that a remote user can damage the host, and second it is the manifestation of a more pervasive minimalist design philosophy (see below).

3. *No hierarchy (compartmentalize)*: There must not be a general notion of something being "more privileged than" something else. Specifically Legion is not more privileged than the objects it supports, and it is completely natural to set up non-overlapping domains/policies. This precludes the notion of a "Legion root," guaranteeing that no single entity can gain system-wide ultimate privileges.

4. *Minimize functionality to minimize threats*: The less one expects Legion to do, the harder it is to corrupt it into doing the wrong thing! Thus, for example we have moved a great deal of functionality into user-definable objects — responsible agents and security agents were discussed here, but similar moves have been made for binding, scheduling, etc. This increases the control that an individual or organization has over their destiny.

5. *If it quacks like a Legion...*: Legion is defined by its behavior, not its code. There are a number of security-related implications of this. First, it's possible for several entities to implement compatible Legion systems; this reduces the possibility of a primordial trojan horse; it also permits competing, warranteed implementations. Second, it opens the possibility of dynamic behavioral checks — imagine a benign worm that periodically checks the behavior of a system that purports to be a Legion, for example.

6. *Firewalls*: It must be possible to restrict the machines on which an object is stored or is executed, and conversely restrict the objects that are stored or executed on a machine. Moreover, the mechanism that achieves this must not be part of Legion. It must be definable on a per class basis just like MayI and Iam. (Of course, like the other security aspects of Legion objects, we expect that the majority of folks will simply inherit this mechanism from a class that they trust).

7. *Punishment vs. Prevention*: It will never be possible to prevent all misdeeds, but it may be possible to detect some of them and make public visible examples of them as a deterrent.

It should be noted that there is an informal, but important link between physical and computer security that is especially relevant to this discussion. Any individual or organization concerned with security must control the physical security of their own equipment; doing this increases the probability that the Legion code at their own site is valid. That, coupled with the security agent's ability to monitor every invocation, can be used to further increase an installation's confidence.

6 Recapping Some Options

This new security model we presented here is to shift the emphasis from "system as enforcer" to user-definable policies — to give users responsibility for their own security — and to provide a model that makes both the conceptual cost and performance cost scale with the security needed. At one extreme, the blithely trusting need do nothing and the implementation can optimize away all the checking cost. At the other extreme, ultimate security suggests staying off the net altogether. Between these extremes:

- High security systems might be willing to accept the base Legion communication mechanism, but not even trust it to MayI or check licenses properly. For these we suggest embedding checks in each member function and use physical security in conjunction with Legion. For example, restrict one's objects to running on only certain, known-to-be-safe computers and accept only a few trustworthy responsible agents.
- Somewhat less sensitive systems might trust the local "imposter checking" mechanisms to adequately ensure that MayI and license checking is done. However, they may still want to invoke MayI on each member function invocation to obtain a high degree of assurance. Such systems may execute authentication protocols with the responsible- and/or calling agent to ensure that the remote Legion is not an imposter.
- In situations where security is not a primary concern, careful systems may feel that a lighter weight check with the "f" function in the license would be appropriate and only call MayI every Nth invocation or after some amount of time has elapsed. Such a system might clear its license cache at random intervals as well.
- And so on.

The point of all this is that there is a rich spectrum of options and costs; the user must choose the level at which they are sufficiently confident. Caveat emptor!

7 Related Work

There is a rich body of research on security that spans a spectrum from the deeply theoretical to the eminently practical, most of which is relevant to this work. In particular, all of the work on cryptographic protocols [11] and on firewalls [18] is directly applicable to the development of Legion itself. Other work, such as that on the definition of access control models [7], on information flow policies[13] and on verification [19] will be more applicable to the development of MayI functions — which we will lean on as we develop a number of bases classes from which users may inherit policies. In the same vein we will lean on existing technologies such as Kerberos [9] Sesame [21], etc.

We are not aware, however, of other work that has turned the problem inside out and placed the responsibility for security enforcement on the user/class-designer. The closest related work is in connection with CORBA; indeed many of the concerns we raised in the introduction are also cited in the *OMG White Paper on Security* [10]. A credo of this work, however, was "no research", and so they retain the model of system as enforcer. Indeed an exemplar of our concern with this approach is where they talk about the trusted computing base (TCB):

"The TCB should be kept to a minimum, but is likely to contain operating system(s), communications software (though note that integrity and confidentiality of data in transit is often above this layer), ORB, object adapters, security services and other object services called by any of the above during a security relevant operation."

It's precisely this sort of very large "minimum" security perimeter that caused us to wonder whether there was another way.

8 Technical Challenges and Future Work

There are many technical issues that we are unable to discuss in depth due to limited space. These issues pose challenging research questions and greatly impact the design of Legion Security. For example,

- *Encryption:* Legion does not specify the use of any particular encryption algorithm. Applications concerned about the privacy of their communication should choose any encryption scheme they deem necessary. But that does raise one question, namely, how much protection of messages should be done by default? Our initial default is to send messages in the clear but digitally signed (with an option to encrypt) — but is that the right performance-cost trade-off?
- *Key management:* Public-key cryptography is the basis of authentication in Legion. However, Legion eschews any distinguished trusted objects. Name and key management thus need to be handled without any centralized component — no single key certification or distribution server. To make the key management simple, we define that every object's unique identifier be the public key of that object. A new key generation scheme is developed to do completely distributed, unique key generation. See [22] for more details on Legion key generation and management.
- *Communication Layer Security:* The mechanisms we have discussed are higher level ones, but we also need to worry about lower level ones — notably the privacy and authenticity of the messages used to implement the higher level concepts. As with the previous mechanisms, we do not want a system-wide policy and we especially want to scale the cost of these mechanisms to the users' needs.
- *Rogue Legionnaires*: Will our users be comfortable enough to use Legion despite the fact that Legion itself

could be corrupted? Do the principles we stated in fact help enough to make users confident? Can we describe the limits of the approach well enough for users to make well-informed decisions?

• *Composition of security policies*: In a multi-policy environment like Legion, what can we say when objects that enforce different policies are used together? In particular what happens when conflicting, even contradictory, security policies operate in conjunction? What can we do to effectively resolve conflict should it arises and help users evaluate combined policies?

• *Expressiveness and Robustness:* Is the model expressive enough to be sufficiently useful to a interestingly large number of users in spite of its limitations? In addition, is it robust enough to engender user confidence? How can we validate the model and effectively demonstrate its expressiveness and robustness?

• There are a host of implementation issues related to other functional aspects of a real system — scheduling, for example — that have security implications (how better to effect denial of service than to simply not schedule the task!).

We will start to test out our ideas and address these questions on a "campus wide" Legion prototype which is currently operational in the University of Virginia. As the overall Legion project proceeds, we will be able to develop the model in a more realistic context and scale.

Our first step, of course, is to complete the model and its operational specification, including choice of protocols, encryption/signature algorithms, distributed key generation, and so on.

A host of base classes with different security policies, such as ACL and Kerberos, will be built in this stage of experiment. Similarly we will implement a small number of login mechanisms — e.g., a password scheme, a question-answer scheme, and a variant of one of these that periodically during a session revalidates the user is still the one that logged in. While these classes will hardly test the limits of expressiveness, they should give us an a handle on the effort required to define familiar policies and will provide an initial selection from which "real users" can derive new classes.

To gain confidence that the design is robust we will be verifying the protocols with tools developed for the purpose here at Virginia. We will also be doing a comparative analysis against other proposals.

The next step is to build "version zero" of the model. We can then compare its performance to distributed systems without security concerns (e.g., PVM, PC++, Mentat) as well as to security services such as KERBEROS.

Several design/build/test cycles are needed to revise and complete our design. An extensive risk analysis is imperative to the procedure. As in other situations, this analysis should guide further efforts to first eliminate failures, and then for those that remain recover and mitigate their effects.

9 Conclusion

The "National Information Infrastructure", NII, will inevitably involve the interaction/cooperation of diverse agents with differing security and integrity requirements. There *will* be "bad actors" in this environment, just as in other facets of life. The problems faced by Legion-like systems will have to be solved in this context.

The model we have posited, we believe, is both a conceptually elegant and a robust solution to these problems. We believe it is fully distributed; it is extensible to new, initially unanticipated types of objects; it supports an indefinite number and range of policies and "login" mechanisms; it permits rational, user-defined trade-offs between security and performance. At the same time, we believe that it has an efficient implementation.

What we need to do now is to test the "we believe" part of the last paragraph.

10 Reference

1. Andrew. S. Grimshaw, William A. Wulf, James C. French, Alfred C.Weaver and Paul F. Reynolds Jr. "Legion: The Next Logical Step Toward a Nationwide Virtual Computer", June 8, 1994. *UVA CS Technical Report CS-94-21*
2. Andrew. S. Grimshaw, "Easy to use object-oriented parallel programming with Mentat", *IEEE computer*, pp 39-51, May 1993
3. Andrew S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,", submitted to *ACM Transactions on Computer Systems*, July 1993
4. Andrew S. Grimshaw, E. A. West, and W. R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, pp. 309-328, vol. 5, issue 4, June 1993
5. Andrew S. Grimshaw, W. T. Strayer, and P. Narayan, "Dynamic OBject-Oriented Parallel Processing" *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 33-47, May, 1993
6. William A. Wulf, Roy Levin, Samuel P. Harbison, "HYDRA/C.mmp: An Experimental Computer System", *McGraw-Hill*, New York, 1981
7. B. W. Lampson, "Protection", *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pp 437-443. March 1971
8. H.H. Hosmer, "The Multipolicy Paradigm for Trusted Systems" *Proceedings of New Security Paradigms Workshop*, pp. 19-32, 1992-1993
9. B. C. Neuman, T. Y. Ts'o, "Kerberos: An Authentication Service for Computer Networks" *IEEE Communications*, Vol. 32, pp. 33-38, Sept. 1994
10. B. Fairthorne, "OMG White Paper on Security", *OMG Security Working Group*, April 1994
11. Bruce Schneier, "Applied Cryptography", *John Wiley & Sons*, INc. 1994

12. Dorothy E. Denning, "A Lattice Model of Secure Information Flow", *Communications of the ACM,* Vol 19, No 5, pp 236-242, May 1976,
13. J. H. Saltzer, "Protection and the Control of Information Sharing in Multics", *Communications of the ACM,* Vol 17, No 7, pp 388-402, July 1974
14. L. Snyder, "Formal Models of Capability-based Protection Systems"IEEE Transactions on Computers, vC-30 n3, March 1981, pp 172-181
15. William R. Cheswick and Steven M. Bellovin, "Firewalls and Internet Security" *Addison-Wesley,* 1994
16. C. Landwehr, "Verifying Security", *Computing Surveys,* Vol.13, No. 3, Sept. 1981
17. G. Benson, Appelbe, I. Akyildiz, "The Hierarchical Model of Distributed System Security", IEEE, May 1988, pp 122-128
18. J. I. Glasgow, G. H. MacEwen, "The Development and Proof of a Formal Specification for a Multilevel Secure System"_*ACM Transactions on Computer Systems,* Vol. 5, No 2, May 1987, pp 151-184
19. R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key cryptosystems", *Communications of ACM,* vol. 21, no. 2 Feb. 1978, pp. 120-126
20. Mark Lomas, Bruce Christianson, "To whom am I Speaking", *Computer,* January 1995
21. Tom Parker and Denis Pinkas, "SESAME Technology Version 3, Overview," http://www.esat.kuleuven.ac.be/cosic/sesame/doc-txt/overview.txt, May 1995
22. Chenxi Wang, Wm A. Wulf, "A Distributed Key Generation Technique". *UVA CS Technical Report, CS-96-08.*