A New Model of Security for Distributed Systems

Wm A. Wulf
Chenxi Wang
Darrell Kienzle

Computer Science Technical Report CS-95-34
University of Virginia
August 10, 1995

Abstract

With the rapid growth of the information age, electronic activities of many kinds are becoming more common. The need for protection and security in this 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 in which the cost scales with the degree of security required. Users ought to be able to select the level of security they need and pay only the overhead necessary to achieve it. 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 tentative implementation plan. A set of technical challenges is 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 simpler systems characterized by a single protected kernel.

However, a large distributed system is usually a federation of distinct administrative domains with separately developed and evaluated security policies. There is no clear notion of a single protected kernel. Instead, a system is comprised of a set of kernels interacting with each other over an open network. This has fundamentally changed the placement of trust and hence the nature 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 distributed 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 "logon" mechanism — some situations will demand a far more rigorous one than others. Moreover we cannot anticipate all the policies or logon 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 [7] 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 next several years will see the wide-spread introduction and use of gigabit wide-area and local area networks. These networks have the potential to transform the way people compute, and more importantly, how they interact and collaborate with one another. The increase in bandwidth will enable the construction of wide area virtual computers, or metasystems, such as have been envisioned both in the computer science community and in science fiction. However, just connecting computers together is insufficient. Without easy-to-use and robust software to simplify the environment the network will be too complex for most users, and will be just so much glass connecting the sites.

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, application adjustable fault-tolerance, improved response time, and greater throughput. Legion is targeted towards wide-area assemblies of workstations, supercomputers, and parallel supercomputers. Legion tackles problems not solved by existing workstation based parallel processing tools such as fault-tolerance, wide area network support, heterogeneity, the lack of a single file name space, protection and security, as well as providing efficient scheduling and comprehensive resource management. At the same time Legion provides parallel processing,

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/

object-interoperability, task scheduling, and security. Our vision is of a system in which a user sits at a Legion workstation and has the illusion of a single very powerful computer on her desk. The user sits down at a terminal[1] and invokes an application on a data set. It is Legion's responsibility to *transparently* schedule application components on processors, manage data transfer and coercion, and provide communication and synchronization. System boundaries, the location of data, and the existence of faults will be invisible.

Legion will consist of workstations, vector supercomputers, and parallel supercomputers connected by local area networks, enterprise-wide networks, and the National Information Infrastructure. The total computation power of such an assembly of machines is enormous, approaching a petaflop; this massive potential is, as yet, unrealized. The potential benefits of Legion include: (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 Legion's Objectives

From our Legion vision we have distilled nine design objectives that are central to the success of the project; site autonomy; an extensible core; scalability, easy-to-use, seamless computational environment; high performance via parallelism; single, persistent namespace; security for both users and resource providers; manage and exploit resource heterogeneity, and fault tolerance.

Site autonomy. Legion will not be a monolithic system. Legion will be composed of diverse resources owned and controlled by an array of different organizations. These organizations, quite properly, insist on having control over their own resources, e.g., specifying how much resource can be used, when it can be used, and who can and cannot use the resource. Therefore, an appropriate model for Legion is that of a cellular automata - where the collective behavior arrises out of the individual behaviors of the components.

Extensible core. We cannot know the future, or all of the many and varied needs of users. Therefore, mechanism and policy must be realized via extensible, replaceable, components. This will permit Legion to evolve over time and allow users to construct their own mechanisms and policies to meet their specific needs.

Scalable architecture. Because Legion will consist of thousands of hosts it must use a scalable software architecture. This implies that there must be no centralized structures, and the system must be totally distributed.

Easy-to-use, seamless computational environment. Legion must mask the complexity of the hardware environment and the complexity of communication and synchronization of parallel processing. Machine boundaries should be invisible to users. As much as possible, compilers, acting

1. We use terminal in its most liberal sense. Terminals may include any human I/O device, glass tty, X interfaces, or virtual reality interfaces such as head-mounted displays and data gloves.

in concert with run-time facilities, must manage the environment for the user.

High performance via parallelism. Legion must support easy-to-use parallel processing with large degrees of parallelism. This includes task and data parallelism and their combinations. Because of the nature of the interconnection network, Legion must be latency tolerant. Further, Legion must be capable of managing hundreds or thousands of processors.

Single, persistent namespace. One of the most significant obstacles to wide area parallel processing is the lack of a single name space for file and data access. The existing multitude of disjoint name spaces makes writing applications that span sites extremely difficult.

Security for users and resource owners. Because we cannot replace existing host operating systems, we cannot significantly strengthen existing operating system protection and security mechanisms. However, we must ensure that existing mechanisms are not weakened by Legion.

Manage and exploit resource heterogeneity. Clearly Legion must support interoperability between heterogeneous components. In addition, Legion will be able to exploit diverse hardware and data resources. Some architectures are better than others at executing particular kinds of code, e.g., vectorizable codes. These affinities, and the costs of exploiting them, must be factored into scheduling decisions and policies.

Fault tolerance. In a system as large as Legion, it is certain that at any given instant several hosts, communication links, and disks will have failed. Thus, dealing with failure and dynamic reconfiguration is a necessity. Current parallel processing systems do not address fault-tolerance. As is the case in the management of heterogeneity there is an extensive literature in reliable distributed computing systems. We will consider two different fault tolerance problems, fault tolerance of Legion itself, and application fault-tolerance.

As well as the design goals above, our design is restricted by two constraints, we cannot replace host operating systems and we cannot legislate changes to the interconnection network.

We cannot replace host operating systems. This restriction is required for two reasons. First, organizations will not permit their machines to be used if their operating systems must be replaced. Operating system replacement would require them to rewrite many of their applications, retrain many of their users, and possibly make them incompatible with other systems in their organization. Our experience with an earlier system, Mentat, indicates that it is sufficient to layer a system on top of an existing host operating system.

We cannot legislate changes to the interconnection network. We must initially assume that the network resources, and the protocols in use, are a given. Much as we must accommodate operating system heterogeneity, we must live with the available system. That is not to say though, that we cannot layer better protocols upon existing protocols, or that we cannot state that performance for a particular application on a particular network will be poor unless the protocol is changed.

In addition to the purely technical issues, there are also political, sociological, and economic ones. These include encouraging the participation of resource-rich centers and the avoidance of the human tendency to free-ride. We intend to discourage such practices by developing and employing accounting policies that encourage good community behavior.

2.2 The Legion Object Model and Philosophy

Legion is an object-oriented system[1]. The principles of the object-oriented paradigm are the foundation for the construction of Legion; we exploit the paradigm's encapsulation and inheritance properties, as well as benefits such as software reuse, fault containment, and reduction in complexity. The need for the paradigm is particularly acute in a system as large and complex as Legion. Other investigators have proposed constructing application libraries and applications for wide-area parallel processing using only low-level message passing services. Use of such tools requires the programmer to address the full complexity of the environment; the difficult problems of managing faults, scheduling, load balancing, etc., are likely to overwhelm all but the best programmers.

Hand-in-hand with our use of 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.

3 The Security Model

In this section we will describe an evolving model that responds to the issues raised in the introduction. Since our research is not yet complete we do not posit this as THE model, but rather an existence proof that there may be a model that resolves the issues.

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. We recognize that in the final analysis some users will not gain the necessary confidence and will simply opt not to use the system; that seems *just right*!

The 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.

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

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 are to invoke 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 member functions (that may be defaulted to NIL); the ones we will describe here are "MayI," "Iam," and "Delegate.".
- user-defined objects can play two security-related roles — those of the "responsible agent", RA, and the "security agent", SA. To play these roles they must provide additional member functions known to Legion; the one described here is "pass".
- every invocation of a member function is performed in an environment consisting of a triple of (unique) object names — those of the operative responsible agent, security 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.

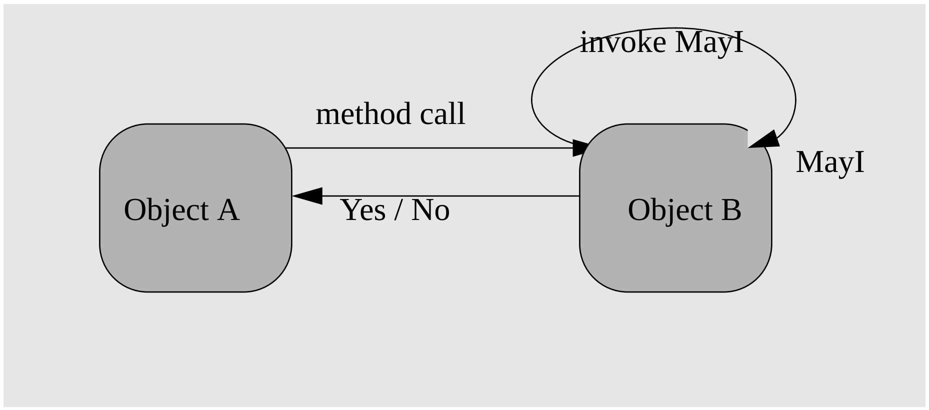
3.1 Protecting Oneself — Privacy

Nobody understands the nature of user requirements better than the users themselves. In Legion, users are asked to do what they are most qualified to do — define suitable security policies that meet their respective needs.

To protect itself, the truly paranoid user’s object can (and should, if they deem it important) include code in every method to determine the identity of the caller and whether that caller has the right to make this call. This cautious user should engage in one of the well known authentication protocols to ensure the caller’s identity.

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 a 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 privacy 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.



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.

3.2 Who is “I”?

The previous discussion left one question unanswered: who or what is the “I” that the MayI function grants access to? Before proceeding to answer this question, let us take a close look at the triple of object names, UIDs, contained in the Legion environment, namely the “calling agent”, CA, the operative “responsible agent”, RA, and the “security agent”, SA.

> The calling agent (CA) is the object that initiated the current method call.
>
> The responsible agent (RA) is a generalization of the “user id” in conventional systems; for the moment its 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.
>
> The security agent (SA) is included to allow security policies to be enforced externally to some collection of objects in question, by determining which objects are permitted to communicate. This enables the implementation of security schemes, such as Multi-Level Security, MLS, where the objects involved cannot be trusted to enforce the policies. It is discussed in more detail in section 4.2 on page 13.

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 any subset of RA, SA and CA. 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 UIDs; 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, security agent, and/or responsible agent (another presumed member function, “Iam”, may be used for this — but of course, it can be defaulted or inherited). Callers, security agents or responsible agents unprepared to adequately authenticate themselves are ipso facto not to be trusted.

As suggested above, the responsible agent will often identify the “user”. But they need not be limited to human “users.” Actually, any Legion object may declare itself to be the current responsible agent should it choose 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).

Consider a scenario where User A, as the current responsible agent, executes “cat myfile.” This command invokes the “read” method on a file object which, after confirming A’s identity, invokes the “read” method on a disk drive object. The disk drive object typically would not recognize the authority of user A to directly access data on the disk — and should thus reject the call! In order to work, the file system must claim responsibility and declare “RA = me” before calling the disk, since the disk object will trust the file system to access it’s data correctly.

The concept of the responsible agent provides a straightforward mechanism for making “logon” a user-definable concept. Typically the first object that a user will invoke will be one that engages in a logon dialog and, if satisfied, declares itself to be the responsible agent (this object most likely also provides access to a few user-specific objects such as directories).

Anyone can define a class with this functionality but, for example, with varying degrees of rigor in the logon dialog (some might require fingerprint or retinal scans, for example, while others might use simple passwords).

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

3.3 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 are 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, SA, CA:             UID;
rights:                 array of boolean;
t:                      time;
n:                      integer;
f:                      function;
cr,cs,cc,ct,cn,cf:      boolean;
```

The "rights" are bits that map one-to-one onto the class'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 cs is true, the environment's security agent must be equal to the license's SA 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
(using 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". So, 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, the default if there is no MayI function defined; caveat emptor!). 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”

If 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.

Note, by the way, that the function f may do anything it wishes, but our intent is that it be the opportunity for an arbitrary, but lighter weight check than might be done by MayI.

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. It follows closely our goal of providing “service sliders” to the users. 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.

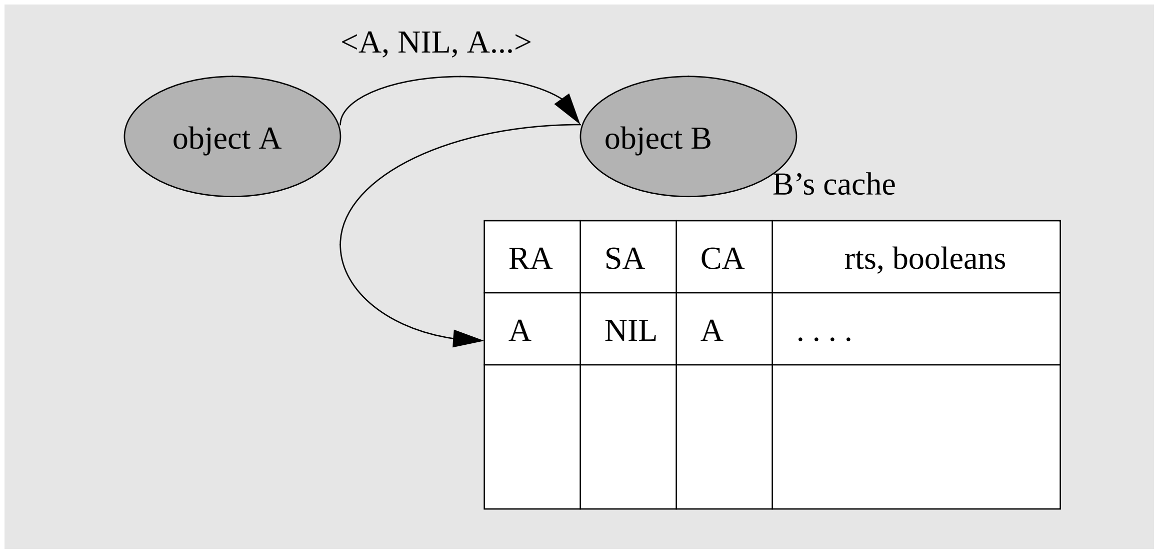
4 Implementation

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, with the licenses indexed by the RA, SA, 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 call<u>ed</u> 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 the situation for an object, A, calling another object, B. Note that this scheme allows the same calling agent to possess different licenses when it operates under different responsible agents, and/or security agents.



There are a number of advantages to this scheme:

- revocation is trivial, and under control of the called object — just delete some (or all) of the licenses in the called object's 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,
- the caller can do no mischief to the license since it does not have write access to it.

4.1 Delegation

In all security models one must consider how rights propagate; for example, if an object A has the right to invoke a method on an object B, is there some way for A to give that right to another object C? Just as the basic privacy 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 UID of the object to which rights should be delegated, and a set of restrictions that limit those rights. For example, consider a user object A, that wants to invoke a compiler, C, and pass it a file, F, with rights restricted to "read only". To accomplish this, A must invoke the "Delegate" method of F to tell it to delegate only the read right to C. 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);
```

Object F, upon receiving the above request, can do whatever it wants, but presumably it will either grant the delegation, refuse it, or grant delegation of a more restricted authority than 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 UID of F.

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

When C attempts to read F,

```
    C: F.read()
```

F’s MayI consults its rights database, recognizes this delegated authority and sanctions the operation. However, if C attempts to invoke any of F’s other methods, F will disallow this.

Since they are defined by the objects themselves, delegation policies can be arbitrarily complex. Classes that want to take extreme precaution 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 time out in much the same manner as a license.

4.2 The Enforcers — Mandatory Policies

As noted earlier, some policies such as MLS cannot depend upon self-policing by the objects involved; we really do need to impose an “enforcer” between the caller and called object. That is the role of the security agent, SA. If the SA field in the environment is NIL, it is ignored and the objects are allowed to communicate freely. Otherwise, the “pass” method of the SA object is invoked with parameters containing the UIDs of the caller, the callee, and the desired method invocation. Thus A’s attempt to invoke B’s method

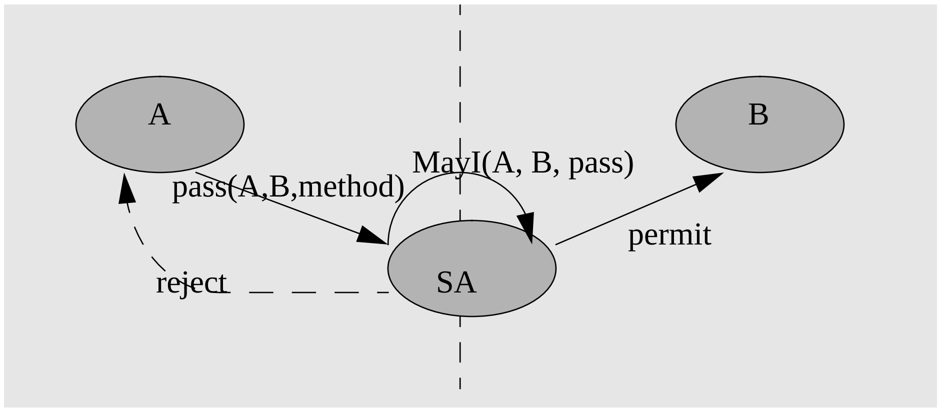
```
    A: B.f(M)
```

M is converted to

```
   A: SA.pass(A, B, f, M)
```

That is, SA is interposed between A and B and gets the opportunity to permit or deny the call. Any object that defines a pass function can declare itself to be the SA, but only if SA is NIL in the current environment (otherwise one could subvert the SA functionality trivially). So, for example, to get access to certain objects one may have to invoke a gatekeeper object that declares itself to be the security agent. This object/agent can then monitor future interactions and enforce whatever policy it wishes. SA can also choose to relinquish its duty if it decides that A and B are trustworthy objects.

One interesting option here is for “pass” to do very little checking. Recall that SA.MayI will be called on the first invocation of “SA.pass” and SA will create a license for this — so by putting the heavyweight checking in its own MayI and setting the license parameters properly, SA can control the overhead to achieve its “comfort level” of security.



Note how the delegation and SA schemes interact with our previously discussed mechanisms and philosophy:

- by default, “Delegate” and SA can be NIL and hence will impose no overhead. And indeed, many classes will favor the default case for performance reasons.
- when “Delegate” or SA is non-NIL, they enforce user-definable policy rather than some global Legion-defined one,
- the “Delegate” and the “pass” function can be as simple or as elaborate as the user feels necessary to achieve their comfort level — the “service slider” approach again.

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 invoke SA, MayI or check licenses properly. For these we suggest embedding checks in each member function. Such a system would certainly want to restrict itself to running on only certain, known-to-be-safe computers and would accept only a few trustworthy responsible agents.
- Somewhat less sensitive systems might trust the local “imposter checking” mechanisms to adequately ensure that SA, MayI and license checking is done, but to ensure privacy will want to invoke MayI on each member function invocation and execute an authentication protocol with the responsible- and/or calling object to ensure that the remote Legion is not

an imposter. Similarly they may want heavy weight mandatory policy checking by SA.mayI on each method invocation.

- In less sensitive situations without mandatory policies, careful systems may feel that a lighter weight check with the “f” function in the license and only call MayI every N^{th} 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 and CORBA Security

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

Our view of the model so far is mostly conceptual and abstract. A host of questions still need to be addressed both on the conceptual and implementation level. For example,

- Can a class designer define “any” policy? The mechanism is still an access control one, after all, so what about the interesting and important information flow policies? Alternatively, is the model expressive enough to be sufficiently useful to a sufficiently large number of users in spite of whatever limitations it has?
- Even if the model is expressive enough, will the implementation be convenient enough to

be used? We waved our hands at the ability to inherit MayI from another class definition, but will it really be that easy? Probably not, since at least some tailoring for the member functions of the new class may be necessary.

- The next level of refinement of the model beyond the description here will be an operational specification; can this specification be made robust enough to engender user confidence? That is, temporarily setting aside the issues of the inevitable bugs and rogues, is the *design* robust? (By contrast we would say that the design of UNIX is brittle since virtually any penetration leads to a total one).
- What about the basic asymmetry in the model? An object may be able to protect itself from use by unauthorized callers, but how does the caller protect itself? In a complex system, the caller may not know what objects are invoked to implement some functionality; what if those objects are bad actors?
- What about the so-called “hookup” problem? That is, what can we say when objects that enforce different policies are used together? In general the combination may be weaker than either alone, so how can we help the user evaluate this. How do we handle moving from the “domain” of one security agent to another, for example?
- At the implementation level, will it be “efficient enough” and will the performance “scale” with perceived increases in security? This leads to a variety of specific implementation questions; for example:

  How much protection of messages should be done by default and how much should control of things like encryption should be given to the user? The issue is, of course, performance related, and our initial default is to send messages in the clear but digitally signed (with a user option to encrypt, of course) — but is that the right trade-off?

- As noted in the introduction, we eschew *any* distinguished trusted objects; that poses a problem for name and key management that we partially resolve by making the UID be the public key of the object it names — but that raises issues of how to do completely distributed, unique key generation.
- What about the issue of rogue Legionnaires? Do the principles we stated in fact help enough to make users confident? Are there other approaches? Can we describe the limits of the approach well enough for users to make informed decisions?
- 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.

Efforts have already been made toward implementing a number of base classes that implement samples of policies, such as ACL, KERBEROS, and MLS. The outcome is rather promising. Similarly in the near future we will implement a small number of logon 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 "logon" agents; 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