A New Model of Security for Metasystems[⋆]

Steve J. Chapin, Chenxi Wang, William A. Wulf, Fritz Knabe, and Andrew Grimshaw

Department of Computer Science, University of Virginia, Charlottesville, VA 22903–2442, {chapin,cw2e,wulf,knabe,grimshaw}@cs.virginia.edu

Abstract

With the rapid growth of high-speed networking and microprocessing power, metasystems have become increasingly popular. The need for protection and security in such environments has never been greater. However, the conventional approach to security, that of enforcing a single system-wide policy, will not work for the large-scale distributed systems we envision. Our new model shifts the emphasis from "system as enforcer" to user-definable policies, making users responsible for the security of their objects.

This security model has been implemented as part of the Legion project. Legion is an object-oriented metacomputing system, with strong support for autonomy. This includes support for per-object, user-defined policies in many areas, including resource management and security. This paper briefly describes the Legion system, presents our security model, and discusses the realization of that model in Legion.

Keywords: security, metasystems

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 a single authority ("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.

[⋆] This work was funded in part by NSF grant CDA9724552, ONR grant N00014-98-1-0454, Northrup-Grumman contract 9729373-00, and DOE contracts DEFG02-96ER25290, SANDIA #LD-9391, and D45900016-3C.

However, in a metasystem 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. We expect it to 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 operating systems.

The intricate nature of metasystems has fundamentally changed the requirements for system security. Within the Legion project, we are investigating a new model of computer security appropriate to large distributed systems.

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. 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 we must assume additional security weaknesses in addition to the usual assumption of insecure communication. We assume that copies of Legion system objects 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. Because Legion cannot replace existing host operating systems, the idea of securing them all is not a feasible option. We 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 one such assumption: that security is absolute; a system is either secure or it is not. A second assumption 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—that burden rests on the owner of the object.

Note that we said that users must feel confident that the privacy and integrity of their data will not be compromised; we did not say that they had to be guaranteed of anything. Security needs to be “good enough” for a particular circumstance, at a cost commensurate with the protection provided. Of course, what is 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 “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 [4] is the archetype of such models.

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 system-wide policy. New policies may emerge all the time, and the complexities of overlapping or 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—at most, we can view it as a set of interfaces, protocols, and agents, some of whom we trust.

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 approach that security is absolute, and everyone either paid the cost of full security or had none, regardless of their individual needs. To succeed, our model must scale along cost—it must have essentially zero cost if no security is needed, and the cost must increase in proportion to the extra confidence one gains. Further, these costs must scale on the basis of individual objects, not only for the system as a whole.

These observations call for a change in our 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, and describe the current implementation of the model within the Legion metasystem. 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

The Legion [3] project at the University of Virginia is an attempt to provide metasystem services that create the illusion of a single virtual machine. This machine provides secure shared object and name spaces, high performance via both task and data parallelism, application adjustable fault tolerance, improved response time, and greater throughput. In all cases, we allow and encourage per-object user-definable resource policies. Legion is targeted towards wide-area assemblies of workstations, supercomputers, and parallel supercomputers. Such a system 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 a metasystem such as 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 applications programmers.

Legion is an object-oriented metasystem. 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 encapsulation and inheritance, as well as providing 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 user's 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." For example, 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, 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 services, or inherit 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 describe the security model and its current implementation in Legion. We first present the design guidelines and principles. We discuss the trade-offs and our design decisions. We then explain how the model works, with particular focus on how it can be used to enforce discretionary policies.

3.1 Design Principles

The Legion Security model is based on three principles:

(i) as in the Hippocratic Oath, do no harm!
(ii) caveat emptor—let the buyer beware.
(iii) 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 model.

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 [12] and is also in some proposed CORBA models [2]. Note, however, that it subsumes more common notions such as protection at the level of file systems. In Legion, files are merely user-defined objects, which 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:

(i) every object provides certain known member functions (that may be defaulted to NIL); we will describe MayI, CanI, Iam, and Delegate.
(ii) there are two objects associated with each operation: a responsible agent (RA) and a calling agent (CA). The RA is someone who can be held accountable for the particular operation. 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. There are a certain set of member functions associated with an RA object. User-defined objects can act as RA by supplying these member functions.
(iii) every invocation of a member function is performed in the context of a certificate which contains the Legion Object ID (LOID) of the RA which generated the certificate, a list of allowed method invocations, and a timeout. The certificate is digitally signed by the maker.
(iv) 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.) at the appropriate time, thus giving objects the responsibility

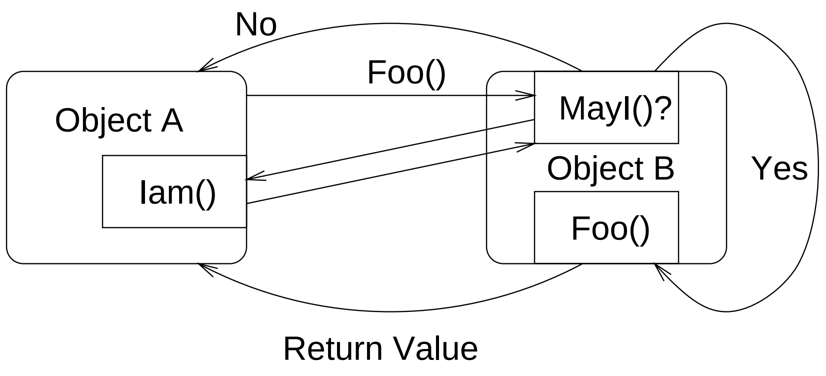

Fig. 1. Object A calls B.foo(), automatically invoking B.MayI()

of defining and ensuring the policy by providing their own implementations of those well-known functions. 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 include code in every method to authenticate the caller and to determine whether that caller has the right to make this call. 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 is correct.

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. Legion automatically calls the MayI function before every member function invocation, and permits that invocation only if MayI sanctions it (see figure 1).

In figure 1, Object A invokes method B.foo. This call is passed to B, and the Legion run-time system automatically invokes B.MayI rather than invoking foo.[1] If B.MayI returns true, then foo is invoked with the arguments passed from A. If not, then an exception is raised and passed back to A (Legion exception handling is beyond the scope of this paper). All the information

[1] Ignore the call to A.Iam for the moment.

necessary to make such a decision (the calling Agent (A), the method being invoked (foo), and the parameters of the call) are available as input to MayI.

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. 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. 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 (the "no play, no pay" option).

3.3 Authentication

The previous discussion finessed one point: who or what is the "I" that the MayI function grants access? 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 certificates. Recall that the certificate contains the object identifier of the responsible agent, and that the calling agent is identified in the method call.

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 the CA, or the RA, or any subset of the two. Indeed, by analogy with familiar systems where "I" is the user, that subset may be just the RA.

Legion makes a specified level of effort to assure the authenticity of the certificate 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. Figure 1 shows a call from B.MayI back to A.Iam to verify A's identity. The specific protocol used between MayI and Iam to authenticate A's identity is immaterial; if Iam satisfies MayI, then the call will proceed, else MayI will fail (and an exception will be raised).

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 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.

3.3.1 Login

The avenue via which Legion users authenticate themselves to Legion is the Login procedure. Login establishes the 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 and delegation.

By the same design principle, Legion does not mandate a single "Login" mechanism. Currently, there is a login object that is 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 generates an additional certificate designating itself as the RA.

There are many advantages to why we shouldn't make this login mechanism universal. For example, logging on to Legion at the University of Virginia may require only a simple password while Legion in the CIA might demand that users 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 (or RA) 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.

3.4 Delegation

In all security models one must consider the question of rights propagation; 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.

In Legion, an object can generate a new certificate to delegate rights, e.g. the user above could generate a certificate granting the bearer the "read" right and pass it to the compiler. If an intermediate object in the call chain wished to delegate rights contained in its current certificate, it could invoke the Delegate function on the RA to generate a more limited certificate.

Our philosophy is that delegation policy is a part of the discretionary policy that should be defined by the object itself. Indeed, delegation policies can be arbitrarily complex or lightweight. Classes that want to take extreme precautions against delegation may choose not to support delegation at all. Alternatively, users can write their own delegation functions or inherit appropriate ones from existing classes.

So far we have discussed three security-related functions: MayI, Iam and Delegate (we defer discussion of CanI to the next section). They are user-defined functions, which together, quite elegantly, 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.

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

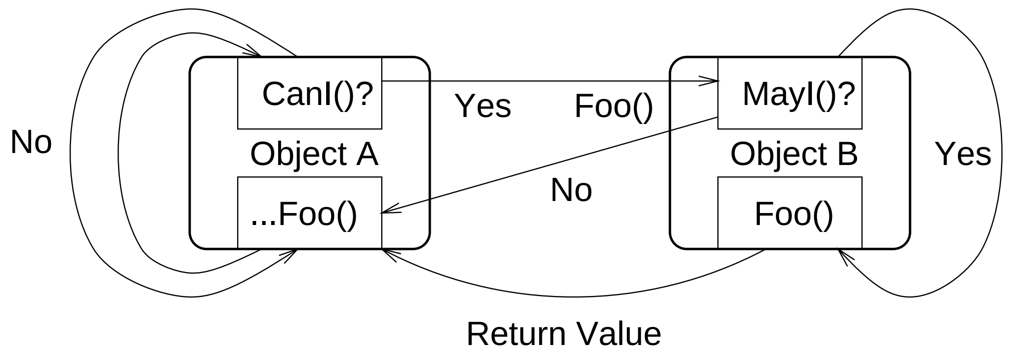

Fig. 2. Automatic invocation of CanI on outgoing calls.

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 performs an outcall to the security agent to verify the validity of each inward call. In this case, we include both compile-time and run-time activities in the actions of the security agent. These may, in fact, be separate but cooperating entities. All 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 CanI 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 CanI methods, the security agent can be ensured that it gets invoked on every call involving one of the objects under its control.

Figure 2 depicts the use of CanI in a method call. Again, Object A invokes B.foo, but the compiler has interposed code so that A.CanI is automatically invoked before the call leaves A. If CanI returns true, then the method call proceeds as in our earlier example.[2] If CanI returns false, an exception is raised.

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 CanI methods are bound into the subclass. Although this seems almost required by

[2] We omit the potential call to Iam to keep the picture legible.

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 our case, if the attack is to be mounted from within an object that the security agent has "wrapped" with its own MayI and CanI 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 attacker's code before deciding whether to allow it into its security regime.

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:

(i) 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.

(ii) 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.

(iii) No privilege 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.

(iv) 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.

(v) 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, guaranteed 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.

(vi) 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). Our prototype implementation uses user-level, per-class and per-host scheduling support to achieve this.

(vii) 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

The Legion security model shifts 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 lie several options, including:

– High security systems might be willing to accept the base Legion communication mechanism, but not even trust it to MayI or check certificates properly. For these we suggest embedding checks in each member function

and use physical security in conjunction with Legion.
– Somewhat less sensitive systems might trust the local "imposter checking" mechanisms to adequately ensure that MayI and certificate 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 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, and not call back to the responsible or calling agent for authentication checks.

Our point 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 [10] and on firewalls [1] is directly applicable to the development of Legion itself. Other work, such as that on the definition of access control models [4], on information flow policies [9] and on verification [7] will be more applicable to the development of MayI functions-which we will lean on as we develop a number of base classes from which users may inherit policies. In the same vein we will lean on existing technologies such as Kerberos [5], RSAREF [8], Sesame [6], 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 [2]. A credo of that 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 affect the design of Legion security. For example,

- Encryption: Legion does not specify the use of any particular encryption algorithm, although our prototype implementation uses the RSAREF public-key encryption library. 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? Should we send messages in the clear but digitally signed? Should we encrypt every message? What is 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 [11] for more details on Legion key generation and management.
- 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 combinatorial policies? How can we express policies to expedite evaluation and composition?
- There are a host of implementation issues related to other functional aspects of a real system—e.g., scheduling—that have security implications (how better to effect denial of service than to simply not schedule the task!).

We are testing out our ideas and starting to address these questions on a Legion prototype which is currently operational both within the University of Virginia and at our research partners (including NSF Supercomputer Centers, DoD MSRC, and DoE National Labs). As the overall Legion project proceeds, we will be able to develop the model in a more realistic context and scale.

We have built several base classes with security policies based on access control lists. We are in the process of incorporating Kerberos authentication into Legion. In cases where our simple login mechanism is deemed insufficient, we are working with our research partners to integrate more stringent mechanisms into Legion.

9 Conclusion

Building metasystems across the Internet will inevitably involve the interaction and 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 developed and implemented, 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.

In the coming months, as we deploy Legion in a nationwide metasystem, we will test the "we believe" part of the last paragraph.

References

[1] William R. Cheswick and Steven M. Bellovin, "Firewalls and Internet Security," Addison-Wesley, 1994.

[2] B. Fairthorne, "OMG White Paper on Security," OMG Security Working Group, April 1994

[3] 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.

[4] B. W. Lampson, "Protection," Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems, pp 437-443. March 1971.

[5] B. C. Neuman, T. Y. Ts'o, "Kerberos: An Authentication Service for Computer Networks," IEEE Communications, Vol. 32, pp. 33-38, Sept. 1994.

[6] Tom Parker and Denis Pinkas, "SESAME Technology Version 3, Overview," http://www.esat.kuleuven.ac.be/cosic/sesame/doc-txt/overview.txt, May 1995.

[7] 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.

[8] RSA Data Security, Inc., http://www/rsa.com.

[9] 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.

[10] Bruce Schneier, "Applied Cryptography," John Wiley & Sons, Inc. 1994.

[11] Chenxi Wang, Wm A. Wulf, "A Distributed Key Generation Technique," UVA CS Technical Report, CS-96-08.

[12] William A. Wulf, Roy Levin, Samuel P. Harbison, "HYDRA/C.mmp: An Experimental Computer System," McGraw-Hill, New York, 1981.