Accountability and Control of Process Creation in the Legion Metasystem[*]

Marty Humphrey
humphrey@cs.virginia.edu
Frederick Knabe
knabe@virginia.edu
Adam Ferrari
ferrari@virginia.edu
Andrew Grimshaw
grimshaw@cs.virginia.edu

Abstract

A metacomputing environment, or metasystem, is a collection of geographically separated resources (people, computers, devices, databases) connected by one or more high-speed networks. The distinguishing feature of a metasystem is middleware that facilitates viewing the collection of resources as a single virtual machine. The traditional requirements of security mechanisms and policies in a single physical host is exacerbated in a metasystem, as the physical resources of the metasystem exist in multiple administrative domains, each with different local security requirements. This paper illustrates how the Legion metasystem both accommodates and augments local security policies specifically with regard to process creation. For example, Legion configurations for local sites with different access control mechanisms such as standard UNIX mechanisms and Kerberos are compared. Through analysis of these configurations, the inherent security trade-offs in each design are derived. These results have practical importance to sites considering any future inclusion of local resources in a global virtual computer.

1 Introduction

The emerging widespread introduction and use of gigabit wide area and local area networks have the potential to transform the way people compute, and more importantly the way they interact and collaborate with one another. The increase in bandwidth will enable the construction of wide area virtual computers, or metasystems (Figure 1). However, in the face of the onrush of hardware, the community has tried to stretch an existing paradigm—interacting autonomous hosts—into a regime for which it was not designed. The challenge is to provide a solid, integrated, conceptual model on which to build applications that unleash the potential of so many diverse resources. The foundation must at least hide the underlying physical infrastructure from users and from the vast majority of programmers. It must support access-, location-, and fault-transparency, enable interoperability of components, support construction of larger integrated components using existing components, scale to millions of autonomous hosts, and provide a *secure* environment for both resource owners and users.

Security services provided by the metacomputer software infrastructure are not unlike traditional services required by uniprocessor operating systems. Users must authenticate themselves

---

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

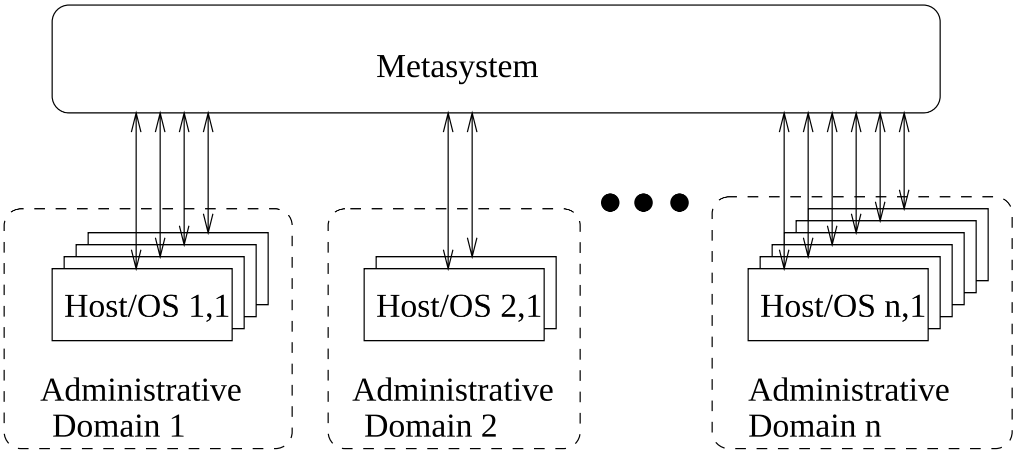

Figure 1: General Metasystem Architecture

to the "system". Users must not be allowed to access arbitrary resources. Logging mechanisms are used to hold users accountable and to track intruders.

However, a solution to the metacomputer security problem requires significant additions over the uniprocessor security solution [3], particularly because the machines that comprise the metacomputer may exist in different administrative domains. In a metacomputing environment, the security problem can be divided into two main concerns:

1. Protecting the metacomputer's high-level resources, services, and users from each other and from possibly corrupted underlying resources (*within* the metasystems layer in Figure 1)
2. Preserving the security policies of the underlying resources that form the foundation of the metacomputer and minimizing their vulnerability to attacks from the metacomputer level (the vertical arrows in Figure 1)

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

The security infrastructure of the Legion metasystem was created to solve these two problems. Legion [5, 6] is an object-based metasystem with a goal of supporting millions of objects spread across thousands of machines. Legion executes as middleware—below the end-user applications but above the operating system (thus, the Legion infrastructure itself is not "privileged code"). Legion provides process management, inter-process communication, persistent storage, a single unified file system, and security services. Legion supports PVM, MPI, C, Fortran, a parallel C++, Java and the CORBA IDL. Legion can select resources for use by applications and securely coordinate large-scale application execution, eliminating the need for the end-user to explicitly log on to each machine, FTP files, create processes, create temporary files, etc. A project of Legion's size and scope is faced with numerous technical challenges, all related to managing the potentially huge number of underlying heterogeneous hardware and software platforms.

In the Legion system, *host objects* represent processing resources. When a Legion object is instantiated, it is a Host Object that actually creates a process to contain the newly activated object. The Host Object thus controls access to its processing resource and can enforce local policies, e.g., ensuring that a user does not consume more processing time than allotted. *Vault objects* in Legion represent stable storage available within the system for containing Object Persistent Representations (OPRs). Just as Host Objects are the managers of active Legion objects, Vault Objects are the managers of inert Legion objects. For example, Vaults are the point of access control to storage resources, and can enforce policies such as file system allocations.

In a metasystem, arbitrary users must not be allowed to allocate resources on arbitrary machines. In contrast, allowing this is analogous to a local sysadmin granting an account to any person without an evaluation of what the person might do on the machine if an account were granted. Sysadmins rarely allow this in order to protect their integrity of the data on their machines and to ensure that resources will be available to "legitimate" users. Similarly, local sites will choose to participate in a metasystem, but contingent on assurances that their local security policies will not be violated.

The specific requirement of Legion that this paper addresses is that Host Objects must only create processes for objects for users that have been authenticated and are authorized to use the underlying resources. Local system administrators will allow their systems to participate in a Legion network (i.e., execute Legion-related processes and store Legion-related objects) only if these processes and files are created according to local policy. For example, a site's security requirements could range from not restrictive (allowing any Legion user to create processes, even if the particular Legion user does not have an account on the computers at the site) to very restrictive (allowing only those users who have personal accounts on the machines on their site, and only after Kerberos authentication). Note that in all cases, only users who have authenticated themselves to Legion are allowed to create processes. Legion must be made to accommodate local policy, over which Legion designers have little or no influence. That is, it is not the intent of the Legion developers to mandate underlying security policy; the challenge in Legion is to adhere to and support local security policies while creating an easy-to-use and secure environment for end users.

The contribution of this paper is that it reveals how this metacomputing problem manifests itself and is solved in the context of Legion. It presents the Legion solution to three site-specific requirements to authentication and authorization for the purposes of process creation. In the first scenario, the site requires standard UNIX access control (similar to most UNIX installations at universities). In the next two scenarios, the sites require Kerberos authentication but differ in their specific security policies regarding process creation *after* Kerberos authentication. The "level of security" as compared to ease of use of each scenario is discussed in terms of the likelihood and ramifications of the compromise of user credentials. The importance of this work is that it describes the solution to real-world security problems that exist as a result of users who increasingly create programs that require transparent, secure access to multiple, geographically dispersed resources. Sites are eager to deploy the Legion metasystem software, but only after receiving assurances that their local security policies are satisfied. Without solutions such as these described in this paper, Legion cannot be deployed on a large-scale basis.

The organization of this paper is as follows. Section 2 presents the Legion mechanisms for security that are independent of any underlying host system. This section covers identity, authentication, authorization, and accountability. Section 3 describes how processes are started and controlled, and how files are written, on a system that employs standard UNIX access control. Section 4 describes two configurations in which Legion integrates with a Kerberized host. Section 5 describes projects that are related to this work. Section 6 contains the conclusions.

2 Security Provided by Legion Mechanisms

The security model for Legion differs significantly from that of conventional systems. A Legion "system" is really a federation of resources from multiple administrative domains, each with its own separately evaluated and enforced security policies. As such, there is no central kernel or trusted code base that can monitor and control all interactions between users and resources. Nor is there the concept of a superuser—no one person or entity controls all of the resources in a Legion system.

Legion programs and objects run on top of host operating systems, in user space. They are thus subject to the policies and administrative control of the local OS. Not surprisingly, the Legion objects running on a particular host must trust that host. This trust does not necessarily extend to objects running elsewhere, however. A critical aspect of Legion security is that the security of the overall Legion system cannot rely on every host being trustworthy. A large Legion system will span multiple trust domains, and even within one trust domain, some of the hosts may be compromised or may even be malicious. For example, two organizations might use Legion to share certain resources in specifically constrained ways. Such sharing would clearly not be acceptable if one organization could subvert the other's objects through its ownership of some part of a Legion system.

The purpose of this section is to provide a discussion of Legion security mechanisms that independent of the underlying security mechanisms and policies of the host systems. The material in this section is presented primarily to establish the context for the remaining sections. For more details regarding this mechanisms, see [3].

2.1 Identity

Identity is fundamental to higher-level security services such as access control. Every Legion object is identified by a unique, multi-field, location-independent Legion Object Identifier, or LOID. One of the LOID fields contains security information including an RSA public key. By including the public key in an object's LOID, it is easy for other objects to encrypt communications to that object or to verify messages signed by it. Objects can just extract the key from the LOID, rather than looking it up in some separate database. By making the key an integral part of an object's name, we eliminate some kinds of public key tampering. An attacker cannot substitute a new key in a known object's id, because if any part of the LOID is altered, including the key, a new LOID is created that will not be recognized by certain core system objects. One drawback in this design is that there is no mechanism for revoking an object's key and issuing a new one, as this step implies a complete change of the object's name.

The default security field of a LOID is more than just an RSA public key. It is actually an X.509 certificate that contains the key. In general, an X.509 certificate pairs a public key with a person's name, organization, identification of the public key algorithm, and other information. A certificate may be signed by a certification authority (CA) that vouches for the association of the key with the identifying information. To cover the case where a recipient doesn't recognize the CA, the CA's own certificate can be chained onto the certificate, allowing the CA's CA to be the basis of authority. The user's X.509 certificate is propagated with requests and method calls made directly or indirectly on behalf of the user. The information in the certificate is used when making entries to access logs.

In a distributed object system such as Legion, the user typically accesses resources indirectly, and objects need to be able to perform actions on his behalf. One way in which to allow intermediate objects to request services on behalf of an originating object is to give the intermediate objects a copy of the private key of the originating object, thus proving authentication. This approach is clearly insecure, as intermediate objects could then maliciously originate operations on false behalf of the originating object. An alternative approach is to have intermediate objects call back to the user or his trusted proxy when they receive access requests in the user's name. This step puts control back in the user's hands. There are several drawbacks to this approach, though. First, the fine-grain control afforded by authorization callbacks may be mostly illusory. It can be very difficult to craft policies for a user proxy (or even the real user himself!) that are much more than "grant all requests"—too much contextual and semantic information is generally missing from the request. Beyond this barrier, callbacks are expensive and do not scale well. In Legion, after all, every object represents a resource of some type, and a callback on every method call would be a crippling performance hit.

The intermediate solution between these approaches is to issue *credentials* to objects. A credential is a list of rights granted by the credential's maker, presumably the user. They can be passed through call chains. When an object requests a resource, it presents the credential to gain access. The resource checks the rights in the credential and who the maker is, and uses that information in deciding to grant access. There are two main types of credentials in Legion: *delegated credentials* and *bearer credentials.* A delegated credential specifies exactly who is granted the listed rights, whereas simple possession of a bearer credential grants the rights listed within it. A Legion credential specifies the period the credential is valid, who is allowed to use the credential, and the rights—which methods may be called on which specific objects or class of objects. The credential also includes the identity of its maker, who digitally signs the complete credential.

2.2 Access Control

Each Legion object is responsible for enforcing its own access control policy. The general model for access control is that each method call received at an object passes through a *MayI* layer before being serviced (see Figure 2). MayI decides whether to grant access according to whatever policy it implements. If access is denied, the object will respond with an appropriate security exception, which the caller can handle any way it sees fit.

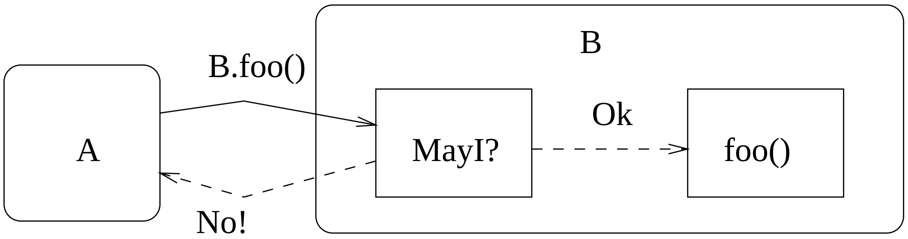

Figure 2: Legion Implementation of Access Control

MayI can be implemented in multiple ways. The trivial MayI layer could just allow all access. Most objects, however, use the Legion-provided default MayI implementation, which essentially defines, on a per-method basis, an *allow* list and a *deny* list. The entries in the lists are the LOIDs of callers that are granted or denied the right to call the particular method. Default allow and deny lists can be specified to cover methods that don't have their own entries.

When a method call is received, the credentials it carries are checked by MayI and compared against the access control lists. For example, in the case of a delegated credential, the caller must have included proof of his identity in the call so that MayI can confirm that the credential applies. Multiple credentials can be carried in a call; checking continues until one provides access.

2.3 Communication between Legion Objects

A method call from one Legion object to another can consist of multiple Legion messages. Because Legion supports dataflow-based method invocation, the various arguments of a method call may flow into the target as messages from several different objects. A message may be sent with no security, in *private mode,* or in *protected mode.* In both private and protected modes, certain key elements of a message (e.g., any contained credentials) are encrypted. In private mode the body of the message is encrypted, whereas in protected mode only a digest is generated to provide an integrity guarantee. Unless private mode is already on, protected mode is selected automatically if a message contains credentials. The mode selected for use by an originating object is applied for all messages indirectly generated as a result of the originating message. For example, a user can select private mode when calling an object. The calls that the object makes on behalf of the user will also use private mode, and so on down the line. Currently, encryption is based on the RSA toolkit (RSAREF 2.0).

In addition to protecting credentials, both protected mode and private mode encrypt a *computation tag* contained in every Legion message, a random number token that is generated for each method call. All the messages that make up a given method call contain the same computation tag. The tag is used to assemble incoming messages from multiple objects into a single method call and to identify the return value for a call made earlier. If an attacker knows the computation tag for a method call, he can forge complete messages containing arguments or return values, even without holding any credentials. The computation tag is treated as a shared secret, and is never transmitted in the clear unless "no security" mode is selected.

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

3 Legion Integration with Standard UNIX Access Control

The previous section described object interactions at the logical level of the metasystem in Figure 1—specifically, how one Legion object can authenticate with another Legion object and exchange secure communication. However, Legion objects must physically exist on a host that is part of the metasystem. This section describes how an object is instantiated on a host that requires only standard UNIX access control.

Our general strategy for isolating Legion objects from one another is to run them in separate accounts on the host system. The accounts that can be used for this purpose fall into two categories:

- For those Legion users who happen to have accounts on the system, objects can run on their normal user accounts.
- For other users, there is a pool of generic accounts that are assigned for Legion use.

The generic accounts usually have minimal permissions. The local Host Object and Vault Object also have their own accounts.

Object creation requests arrive at the Host Object as normal method invocations, and can thus be controlled using the standard Legion access control mechanism for methods. For each request, the host checks the credentials against the user LOIDs and groups that are allowed to create objects on it. If everything is acceptable, it next selects an account for the new object to run in; depending

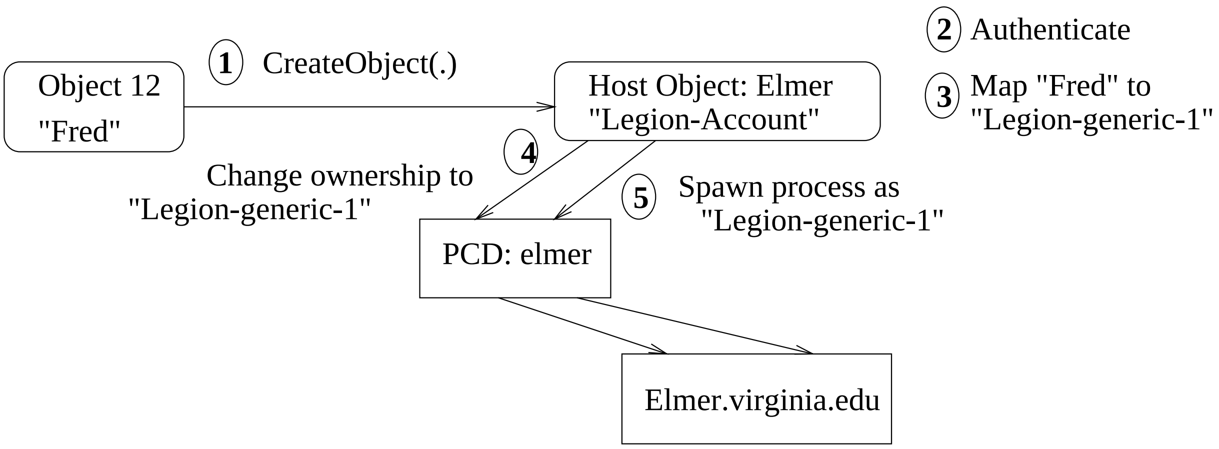

Figure 3: Object Creation on a Standard UNIX Access Control Host

on the credentials in the creation request and its local configuration, it may choose a local user account or one of the generic accounts. The accounts are subject to scheduling and resource control just like CPU time, memory usage, and so on; an object's lease on an account, especially a generic account, is usually limited.

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

Finally, the host needs to spawn the actual process that will execute the object on the appropriate account. To carry out this step, and to change ownership of the object's persistent state, the host requires access to some privileged operations. However, the host does not execute with root permissions. Access to these required privileged operations is encapsulated in a *process control daemon* (PCD) that executes on the host, providing services to the Host Object in a controlled fashion. The PCD is a small, easily vetted program that runs with root permissions. It is configured only to allow access by the host account. Two of its key functions are to permit changing directory ownership and to create new processes on a designated account. The PCD limits the accounts for which this can be done to a set configured by the local system administrator. The set includes the generic Legion accounts and potentially the accounts of local Legion users.

A simplified example of this operation is shown in Figure 3. In this example, *Object 12*, which is executing on Fred's account on some machine in the metasystem, and has access to Fred's Legion credentials, wants to create an object on the host *Elmer.virginia.edu*. To do this, *Object 12* asks the Host Object executing on Elmer to start a process for the new object. First, the Host Object confirms that *Object 12* is acting on Fred's behalf by looking at the credentials contained in the request. Then, the Host Object maps a request by Fred to a generic account (*Legion-generic-1*) on *Elmer.virginia.edu* that has been established at the time that Legion was installed on *Elmer.virginia.edu*. Finally, the Host Object asks the PCD to spawn a new process as *Legion-generic-1* for the new object.

As the PCD starts the object running, the host logs an audit trail using the X.509 information for the user whose credentials accompanied the request. The audit trail provides essential information if the new object misuses local resources. If the object has exceeded its use of local resources, the host can request that the PCD kill it directly. When an object loses or relinquishes its use of an account, the Host Object uses the PCD to change the ownership of its persistent state back to the Vault Object. If the object is reactivated later on a different account, ownership of the state can be changed to the appropriate user-id. After an account is reclaimed, the PCD terminates all processes running on it and generally cleans it up.

Security Analysis. In accepting this approach, a sysadmin at a local site is trusting the Legion software to legitimately map Legion identities to local accounts (if the PCD is configured to map to non-generic local accounts). If the Legion credentials for a particular user are stolen, the risk to the system is less when configured for generic accounts than with non-generic accounts (by their nature, when a user is finished with a generic account, no persistent state remains). Clearly, a negative aspect of this approach is that a site must install the PCD as privileged code, creating a potential point of attack for intruders. However, this code has been vetted by numerous experts, increasing the confidence on the part of local site regarding the safety of this code. Of course, if the Legion Host Object account ("Legion-Account" in Figure 3) is cracked, the intruder can create processes under the accounts of any local Legion users.

Overall. The PCD-based implementation is sufficient for many local system administrators. Legion authentication is used to determine who gains access to local resources, and the resources made available are also constrained to those usable from a limited set of accounts. Detailed logging provides accountability. The safety of credentials is a chief design goal in the security architecture and mechanisms of Legion. An alternative, simple approach is to have all Legion objects execute under the "Legion-Account" account. In general, we have found that sysadmins do not like this approach because of limited accountability—as far as they see, only one account "does anything" with regard to Legion. We (the Legion designers) do not advocate this approach, because it does not provide the necessary isolation, as all files and processes are owned by one oner id (meaning that one Legion user can use UNIX mechanisms to destroy or subvert another Legion user on the same host).

4 Legion Integration with a Kerberized Host

Increased security concerns have caused many sites to switch from standard UNIX access control to the use of Kerberos [9]. Kerberos is a trusted third-party authentication, in which users and services register their keys. In this paper, familiarity with the basic Kerberos protocols are assumed.

It is important to understand that the "Kerberized Host" in this section refers to a host that is executing the MIT source code distribution [7]. This paper does not discuss efforts to integrate Kerberos directly with public key cryptography [10], because this paper focuses on integration with widely-deployed Kerberos systems. Similarly, the use of Proxiable tickets in Kerberos is not discussed, because their usage is not widely supported. While its support is not directly discussed, much of the discussion is applicable to AFS.

In this section, assume that a simple Legion metacomputer is being constructed. There are only two machines involved:

Khost The Kerberized Host machine

NKhost A machine that does not require Kerberos authentication (instead, it uses only password-based authentication)

Additionally, it is useful to define the following entities:

L-creds credentials that are necessary to function in the Legion virtual computer

K-creds Kerberos credentials; obtained via Kerberos **kinit** either explicitly or implicitly

admin Legion user with some administrative duties; ″admin″ does not have an account on either Khost or NKhost; has L-creds; *may* have K-creds (if admin logged into Legion virtual machine via some account on NKhost, then does not have K-creds; if via Khost, it has K-creds, though not as ″admin″, because ″admin″ does not exist as a Kerberos principal on Khost)

Legion-Khost Kerberos principal with account on Khost solely to execute processes related to Legion virtual computer; has K-creds; does *not* have L-creds (there is no Legion user named ″Legion-Khost″)

Alice-KL has an account on Khost and wishes to participate in Legion virtual machine executing on Khost; has K-creds; has L-creds; has an account on NKhost

Bob-L has an ″account″ on Legion virtual machine but no account on Khost; has L-creds; does *not* have K-creds; has a UNIX account on NKhost

4.1 Kerberos Background: .k5login and .k5users

In Kerberos, there is the capability to allow one principal to grant access to another principal (after the other principal has authenticated). The file **.k5login** allows one user to unconditionally grant another user the ability to spawn processes as the first user. For example, if˜bob/.k5login contained ″jim″, jim could ″ksu bob″ and have a running shell whereas new processes are tagged as being owned by bob. Note that in this case, jim *does not* acquire bob's credentials; rather, in this case, jim, executing as UID bob, has a copy of the Kerberos credentials jim had immediately prior to executing **ksu**. **.k5users** is more restrictive than **.k5login**; **.k5users** lets bob allow jim to execute only certain binaries as bob, for instance ″/bin/ls″. The entry in˜bob/.k5users in this case is ″jim /bin/ls″. In this case, jim *cannot* execute a shell as bob; jim could only ″ksu bob -e /bin/ls ″.

4.2 Kerberos Solution #1: k5login

A simple solution to allow a user such as Alice-KL with an account on Khost to access the Legion virtual machine is to add “Legion-Khost” to her **.k5login** file. This approach allows Legion-Khost to execute ″binary1″ as Alice-KL by invoking ″ksu Alice-KL -e full-path-to-binary1″. For Alice-KL to start a process on Khost, essentially the same steps as in Figure 3 are taken. The difference in this situation is that the PCD does not exist, as the Host Object can create processes directly as Alice-KL via invocations of **ksu**. In this configuration, Legion mechanisms will ensure that Bob-L will *not* be able to start new processes on Khost, but Bob-L will be able to use services (processes) of Alice-KL, but only if Alice-KL has configured Legion authorization mechanisms to let Bob-L.

Security Analysis. The analysis consists of a number of cases:

If the Legion-Khost account is compromised (i.e., an attacker obtains Legion-Khost Kerberos credential cache, or breaks into the Host Object and, for example, causes the Host Object to execute a binary that allows the credential cache to be read), then *all* Legion users on Khost, even if they have never used Legion, have been compromised. The attacker cannot get their K-creds, but can start processes on their accounts[1]. There is a variation of this approach, beyond the scope of this paper, that uses **.k5users** instead of **.k5login** in order to reduce the scope of attack is the Legion-Khost is compromised. However, this approach is substantially more complicated to implement and analyze.

If Alice-KL's K-creds are stolen (i.e., either by some activity irrespective of Legion, or if Alice-KL gives her K-creds to a Legion "con-artist" object), then only Alice-KL's account is compromised. In this case, an attacker can replace legitimate Alice-KL service with corrupted service, thus tricking Bob-L if Bob-L wants to use the service.

If Alice-KL's L-creds are stolen Alice-KL's account is compromised because the Host Object can be asked to start jobs on her account.

If admin's L-creds are stolen an attacker can effectively shut down the Host Object but not break security. Note: admin has no special privileges with regard to the host object, beyond being able to change the ACLs on the Host Object.

Overall. The fact that Alice-KL can start a process on the Khost without directly obtaining and presenting K-creds is both positive and negative: The use of the Legion virtual machine by Alice-KL is easier and perhaps more secure because she does not need to directly acquire Kerberos credentials. A potential problem is that the Legion-Khost has unlimited access to the Khost account of Alice-KL. For this reason, Alice-KL and/or the sysadmins of Khost might require that Alice-KL get a separate account on Khost for use with the Legion virtual machine. For many installations, this approach is sufficient, satisfying the requirement of Kerberos authentication before use of the physical resources (the authentication in this case is by the Legion-Khost principal). A sense of security is provided by the Legion mechanisms based on L-creds that have timeouts, recovery mechanisms, and potentially very specific scope and privilege. It is also very easy to implement.

4.3 Kerberos Solution #2: KProxy Object

In general, a problem of the **k5login** approach is that a user must grant unlimited access to her account by the Legion-Khost principal. A second problem is that a user such as Alice-KL (or more precisely a person *impersonating* Alice-KL) does not have to authenticate to the KDC of the Khost Kerberos realm in order to use the physical resources. A second approach eliminates these problems, but at the cost of simplicity.

The essential component of the design is a Legion KProxy Object for each user. This KProxy Object securely holds the Legion user's Kerberos credentials. The KProxy Object for user Fred executes under Fred's uid on a machine upon which Fred has an account. Whenever a Host Object anywhere in the metasystem wants to create an object on Fred's behalf on its associated physical machine, the Host Object performs a call back to the KProxy Object for Fred to obtain a valid ticket for that particular host. Fred's KProxy Object will only issue Fred's Kerberos credentials if Fred's valid Legion credentials are presented in the request (more generally, the access control

---

[1] The attacker can get their K-creds if the user is logged in.

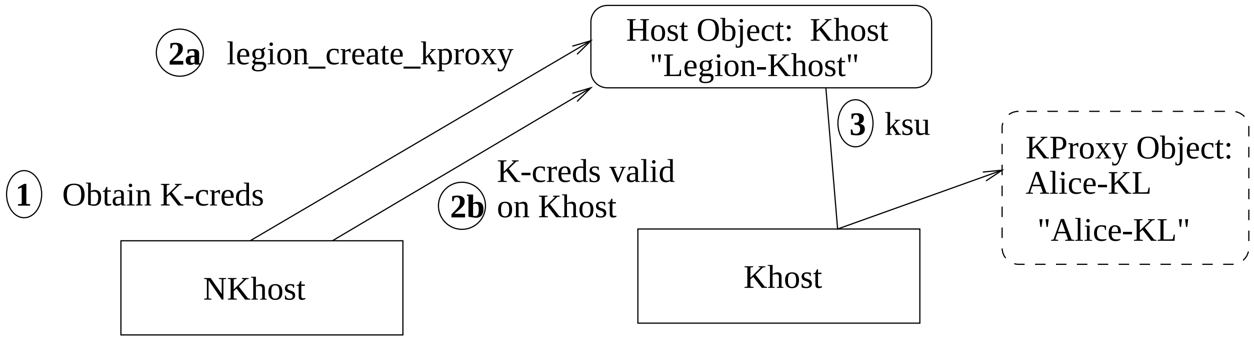

Figure 4: Object Creation on Kerberos-Controlled Host

mechanisms of Fred's KProxy Object can be configured to issue Fred's Kerberos credentials to any object that presents valid credentials on behalf of any user with whom Fred has previous established a trust relationship). The Host Object creates the new object via a call to **ksu**, without requiring the use of the **.k5login** file.

Note that all communication of secret information is either done via Kerberos mechanisms (DES) or Legion mechanisms (RSAREF). Cross-realm authentication is immediately and transparently supported in this design: **kinit** only has to be performed once for each group of Kerberos realms that support cross-realm authentication with each other. The Legion KProxy object will automatically obtain TGTs for the other realms based on the existence of a valid TGT for a given host.

The steps that Alice-KL must take in order to create her KProxy object are shown in Figure 4. First, Alice-KL obtain her K-creds from machine NKhost (this can also be performed on Khost, although it doesn't have to be). By default, these K-creds are tied to the IP address of NKhost (if not—if the KDC supports **kinit -A**— many aspects of this approach are simplified). On NKhost, Alice-KL then executes **legion_create_kproxy**, which asks the Host Object on Khost to create an instance of *KProxy_class* on Khost. As part of this (not shown), Alice-KL interacts with the KDC to obtain a ticket that is usable from Khost. These K-creds are then used by the Host Object in an invocation of **ksu** to actually *create* the KProxy object that will hold Alice-KL's Kcreds. This KProxy object will execute on this machine under Alice-KL's account. Note that neither Alice-KL's **.k5login** nor **.k5users** contains an entry that directly allows Legion-Khost to execute this **ksu**; instead, **ksu** is invoked with an explicit copy of Alice-KL's K-creds. Now, anytime in the future that the Host Object on Khost wants to create an object on Alice-KL's behalf, it interacts with this KProxy object to obtain a valid ticket for use in the **ksu** invocation. Additionally, any Host Object in the realm will interact with Alice-KL's KProxy object when creating objects.

Why can't the Legion user obtain K-creds for a particular computation *before* starting the computation, thus eliminating the need for the KProxy object? Legion is an object-based system in which the necessary functionality to start and coordinate large-scale computation may be spread across hundred or thousands of objects. When a user *originates* computation, the user has no idea on what machines processes ultimately will be started either directly or indirectly on behalf of this user request. For example, the user may attempt to ″run discreet-simulation-1 1000 times″. At this point, the user has no idea (he may not care) which machines are then selected by scheduler objects. Thus, when the user originates computation, he cannot obtain tickets (which are tied to IP addresses) for all of the machines on which processes will be started. Our approach uses a call back to the KProxy object to get the appropriate ticket for each machine.

A second, related note is that the user may not be present (i.e, logged onto a machine in the Kerberos realm) when the Legion software attempts to spawn processes on his behalf. For example, assume that the user starts a long-running activity on one machine, and then logs off. The Legion software might then need to spawn a process on another machine (if, for example, the computation is pipelined, where each element of the pipeline is a sufficiently long computation). Again, we need a call back to something (the KProxy object in our approach) in order to obtain the necessary ticket. One approach might be to somehow get a ticket from the process that already exists that is performing the active part of the pipelined computation (because the credentials cache is present for this process, and presumably it contains a TGT). This is essentially the approach of the KProxy object.

The key limitation that this approach overcomes is that a process can only be started on Khost as user Alice-KL if Alice-KL has previously authenticated herself to the KDC of the Kerberos realm. In addition, at any point, Alice-KL can control the creation of processes under her account by either limiting the lifetime of the ticket held by her KProxy Object, or eliminate her KProxy object completely. However, it is still the case that the Legion user must trust the Legion software to create processes only that she intended.

Security Analysis. The analysis consists of a number of cases:

If the Legion-Khost account is compromised If Alice-KL uses the subverted Khost, the Host Object can use her L-creds to obtain the K-creds from the KProxy. The K-creds can then misused by the Host Object, for example, to spawn arbitrary processes under Alice-KL's user id (such as "rm *"). Note that unlike the **k5login** approach, where a subverted Khost Legion account can immediately abuse all local Legion user accounts, this approach limits the attacker to misusing the accounts of users currently starting objects on the Khost. This allows intrusion detection as an approach for limiting the damage caused by an attacker.

If Alice-KL's K-creds are stolen The same ramifications as the **k5login** approach.

If Alice-KL's L-creds are stolen The same ramifications as the **k5login** approach.

If admin's L-creds are stolen The same ramifications as the **k5login** approach.

Overall. This approach trades off some additional complexity in terms of the systems structure and some extra effort on the part of users (who no longer get to execute just a single command to login) for an added measure of attack containment. This approach meets the requirement that the user actively authenticate through the Kerberos mechanism before using the local resources (unlike Kerberos **k5login** approach). This approach incurs added overhead due to the call-backs to the KProxy object; however, these call-backs only occur at the time of object creation, so the impact should not be significant.

5 Related Work

There are several projects being conducted to support inter-operability between a particular security infrastructure and Kerberos, for example supporting a single login for NetWare and Kerberos [1]. This project has similar goals to Legion's integration with Kerberos—in particular, no changes to Kerberos and no reduction in security in either security realm due to the single login. A significant difference between the Legion project and these projects is that Legion attempts to build security mechanisms the can be viewed as being *on top of* underlying security mechanisms of host systems, whereas these projects generally attempt to support single sign-on of co-existing realms.

Minsky and Ungureanu address the need of unifying heterogeneous security policies in distributed systems by introducing a formalism that describes various security policies [8]. A unified mechanism is used to enforce the security schemes. This work is important for the construction and analysis of security policies in Legion, in that the Legion mechanism must support a wide variety of local policies. However, in Legion, the approach is that the local sites can have whatever security policy they want, and it is very likely that it will not be specified formally. Requiring every local site to specify their security policies in a single formalism is difficult if not impossible, severely impeding Legion's deployment.

Yialelis and Sloman describe a security framework for object-based distributed systems [12]. This project is related to the work in Legion, because it attempts to allow the development of secure distributed applications on operating systems with varying degrees of security mechanisms built it. While this work is similar to the Legion mechanisms described in Section 2, the work of Yialelis and Sloman is CORBA-based and does not address the general metasystems requirements, such as hardware heterogeneity and multiple administrative domains.

Globus [4] is another metasystem research project, and as such is addressing many of the same issues as Legion. In many instances, convergent evolution has led to similar solutions to these problems. For example, Globus has a small, easily-verified module called the Gatekeeper that runs as root and is responsible for remote process management, in much the same manner as the PCD. The manner in which Globus integrates with Kerberos is through use of the Generic Security Services API (GSS-API [11]). The level of granularity of the GSS-API and the Legion object model are fundamentally different: In GSS-API, two applications such as FTP and FTPD establish a security context and then communicate based on the security context. In Legion, objects are significantly more fine-grained than applications such as FTP and FTPD—the overhead to establish contexts establishing and deleting security contexts for each pair of communicating objects is intuitively too expensive, as the number of object-object communications is potentially quite large in the life of a computation.

CRISIS [2] is the security architecture for the WebOS project at UC Berkeley. The WebOS provides many of the same high-level services as Legion. WebOS is fundamentally different than Legion in that while WebOS focuses on system-level support for building and running wide-area applications, Legion's goals are to provide an object-based programming model suitable for such a wide-area application. The principle goals of CRISIS are similar to the goals of the security architecture in Legion: to use redundancy to reduce the likelihood of system compromise, cache whenever possible to improve performance, support fine-grained control over delegated rights, make extensive use of logging, support local autonomy, and to make the design as simple as possible. A difference between the two systems is a result of Legion's support for autonomy which is not a focus of WebOS: Legion supports dynamically-configured local security mechanisms, and CRISIS supports uniform mechanism across all of the nodes of the wide-area system (although policy within each node may be separately defined).

6 Conclusions

We have introduced the Legion security model, and within that model we have presented three alternatives for adapting Legion to local access control and authentication policies. On a practical level, these designs are important because they are in active use at various local sites within deployed Legion networks. Therefore, the security implications of each are of great interest to system administrators and users at such sites. On a more general level, these designs demonstrate the degree to which the Legion architecture can accommodate and adapt to site-local requirements.

The Legion system is currently widely deployed, incorporating diverse resources at Supercomputing Centers, Labs, and Universities. For more information about the current status of the Legion system, see http://legion.virginia.edu. The power of the environment increases with the scale and scope of the system. We continue to actively integrate new sites into Legion. A natural part of the evolution requires us to adapt the Legion security architecture to new site-local policies and mechanisms. The work presented here describes our current dominant site configurations. In the future we expect to see this set expand as Legion deployment increases.

References

[1] William A. Adamson, Jim Rees, and Peter Honeyman. Joining security realms: A single login for netware and kerberos. In *Fifth USENIX Security Symposium*, June 1995.

[2] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. CRISIS: A wide area security architecture. In *Seventh USENIX Security Symposium*, January 1998.

[3] Adam Ferrari, Frederick Knabe, Marty Humphrey, SteveChapin, and Andrew Grimshaw. A flexible security system for metacomputing environments. Technical Report CS-98-36, Department of Computer Science, University of Virginia, Charlottesville, Virginia, December 1998.

[4] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *Fifth ACM Conference on Computers and Communications Security*, November 1998.

[5] Andrew S. Grimshaw and William A. Wulf. Legion: A view from 50,000 feet. In *Fifth IEEE Symposium on High Performance Distributed Computing*, August 1996.

[6] Andrew S. Grimshaw and William A. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[7] Massachusetts Institute of Technology Kerberos Team. Kerberos 5 Release 1.0.5. http://web.mit.edu/kerberos/www/.

[8] Naftaly H. Minsky and Victoria Ungureaunu. Unified support for heterogeneous security policies in distributed systems. In *Seventh USENIX Security Symposium*, January 1998.

[9] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[10] M.A. Sirbu and J.C.-I. Chuang. Distributed authentication in kerberos using public key cryptography. In *1997 Symposium on Network and Distributed Systems Security (SNDSS'97)*, February 1997.

[11] J. Wray. Generic security service application program interface, version 2. RFC 2078, January 1997.

[12] N. Yialelis and M. Sloman. A security framework supporting domain-based access control in distributed systems. In *1996 Symposium on Network and Distributed Systems Security (SNDSS'96)*, February 1996.