The No-Policy Paradigm: Towards a Policy-Free Protocol Supporting a Secure X Window System

Mark Smith
AT&T Bell Laboratories
Guilford Center
Greensboro, North Carolina 27420

Abstract

This paper proposes a framework for a secure, interoperable X Window System* protocol. It reintroduces the concept of a *policy-free* protocol within the context of the X Window System with the goal of achieving industry consensus on that protocol for secure operation. We claim that this consensus can be achieved without requiring vendors to agree on a single standard security policy, much less agreeing on a particular implementation of a security policy. A policy-free protocol framework and its impact on X Window applications is proposed. The relevance of this framework to other trusted systems is explored.

1 Introduction

The problem of constructing a secure X Window system has been treated in several prior accounts [2, 4, 6, 11]. Epstein [4] in particular contains a breakdown of the secure X Window problem by problem area (e.g., mandatory access control, discretionary access control, object reuse) and by level of trust (e.g., B1 trust, Compartmented Mode Workstation trust, B3 trust). These treatments hint at the problem of creating a *secure interoperable* X. In order for any X Window system to be consistent with X's original design goals, it must be interoperable. That is, it must be possible for the user to run an *X Window client* from any machine on the network where the X Server runs, independent of the hardware architecture of the machine the client is running on. See Figure 1 for a possible X Window topology.

Interoperability is attained through the specification of a standard X protocol as defined by the MIT X Consortium. As vendors gain more experience with X and desire additional X capabilities, the standard evolves. There is a facility called *extension* that allows a vendor to "burn in" novel X protocol extensions. These vendor-specific extensions may be reviewed by the MIT X Consortium for eventual inclusion into a new X standard. When an extension is approved and included in the X Window standard, the facilities it provides become effectively interoperable. The MIT X Consortium also includes in its "sample server" a set of extensions which are not yet in the standard but which are deemed to be of sufficient value to warrant inclusion. A vendor need not (and sometimes does not) pass on the MIT sample server verbatim to its customers. Instead, a vendor may choose to select or reject any extensions it receives from the MIT sample server. An extension thus becomes effectively interoperable if all vendors choose to include it in their delivered X Servers.

There are two problems with X Window security extensions.

1. The MIT X Consortium has historically included only the most minimal notion of access control in its X protocol standard.

2. Several vendors have added access control extensions to the X protocol. These extensions are not interoperable; moreover, they do not necessarily reflect the same security policy.

We propose a framework for the creation of a single X protocol extension that is capable of supporting all the security features, attributes and policies that vendors (and their customers) desire. Industry consensus on a protocol fitting this framework would yield an X Window extension, leading eventually to an interoperable secure X Window system.

The proposal defines separable *policy-free* and *policy-defining* subsystems. This rearchitecture of security-providing facilities is similar to that proposed

*The X Window System is a trademark of the Massachusetts Institute of Technology.

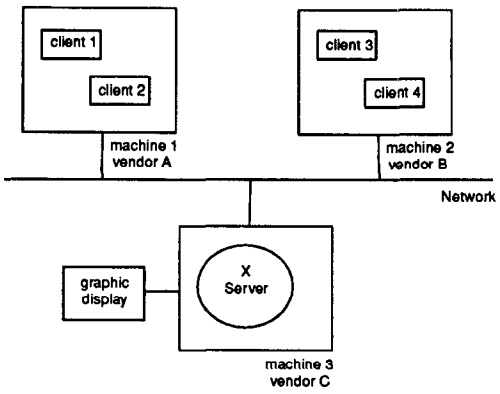

Figure 1: An X window topology

in [1], [3] and suggested by [12]. In particular, [1] suggests a policy-free mechanism for access control with architectural advantages similar to those of our proposal. Perhaps more relevant is the fact that X was originally designed based on the principle of a policy-free protocol [7]. X is *window management policy-free*, allowing vendors the freedom to design and develop X window managers however they see fit. Though there has been some criticism of policy-free protocols for use in a graphical system, they have the distinct advantage of allowing vendors to standardize on relatively non-controversial, mechanistic protocols, rather than on much more controversial window management policies, for example.

Experience has shown that similar difficulties exist when vendors attempt to standardize on a particular protocol supporting security attributes or policies. Even in cases where vendors agree on the security policy (e.g., the Compartmented Mode Workstation policy), they do not necessarily agree upon the protocol, since security policies can be implemented using various techniques (e.g., separation vs. fine grained access control) and typically these techniques impact the protocol. Furthermore, certain aspects of "standard" security policies may be left to the interpretation of the vendor and the accrediting body, and different "sub-policies" may be allowable under the standard (for example, access control lists are optional for the Compartmented Mode Workstation). Finally, a single vendor may desire to support a customer base demanding different security policies.

What is desired therefore is a protocol that passes two tests: (1) it must allow the vendor to use whatever technique the vendor desires to implement the security policy, and (2) it must allow implementation of the security policy of the vendor's choice. It is also highly desirable that the protocol minimally impact the performance of the system. We claim that only a protocol passing these tests has a chance to attain vendor consensus leading to interoperability.

A framework for creating such a policy-free protocol is proposed in the following section. The protocol's impact on the X Server and policy-defining X clients are explored, followed by a summary of the expected impact on the embedded base of X Window clients (referred to as the COTS, or commercial off the shelf, client base). Finally, a generalization of the architectural principle of policy-free interfaces supporting security policies is discussed.

Throughout this paper, the term "X" refers to the MIT X Window System, "CMW" refers to the Compartmented Mode Workstation, and "ACI" refers to Access Control Information.

2 Proposed Protocol Framework

2.1 Protocol Overview

The attempt to specify a protocol supporting various security policies and implementations thereof is similar to the functional decomposition methodology of an object-oriented system, or more particularly that of a strongly typed system. This decomposition is thus similar to the one described by LOCK [10], which specifies security policy based on object and subject type rules.

In particular, "subjects" and "objects" should be type-classified in a policy-free protocol in order to support a large class of labeling security policies. However, such a classification is not sufficient to specify very fine-grained policies that make use of object access control lists, or subject capabilities [12]. Therefore, for the protocol to be sufficiently general, it

must also include the ability to identify ACI on a per-subject or per-object basis.

There are other issues to consider besides what might be called "pure" access control. The protocol must not preclude the construction of a secure infrastructure capable of being accredited. Such an infrastructure must pass certain integrity tests; for example, a change to an object's ACI cannot be made while it is being accessed (the *tranquility* property). Also, provision must be made for trusted paths, so that X clients doing trusted I/O have provably unspoofable access to the physical display, and so that the X Server has unspoofable access to trusted policy-cognizant X clients. Other infrastructure requirements will be introduced in the following sections as well.

2.2 Architecture

The key aspect of the architecture shown in Figure 2 is the construction of a *policy-defining client* (PDC), providing the basic Access Control Decision Function (ADF) as defined in [3] and [12]; the X Server, in nearly all cases, provides the analogous Access Control Enforcement Function (AEF). Note that the PDC has the same client/server relationship to the X Server as any other client; it can be similarly relocated to any machine on the network.

In addition to the server/client relationship, the network over which the X Server, PDC, and other X clients communicate must provide certain authentication capabilities. In particular, it must not allow an untrusted application to spoof the PDC, and it must allow the PDC to authenticate any other client. This particular requirement also implies that the traditional meaning of "X client" must be strengthened: historically, "X client" has meant "a connection to the X Server." In theory, the X protocol allows more than one process to communicate on a single connection to the X Server, although in practice this capability has not been widely used. In order for the PDC to authenticate a client, it must assume that a client is a subject, and therefore there can only be a single identifiable client on a connection. The network must provide a way to restrict access to a connection in the required manner.

2.3 Proposed Mechanisms

The framework for the policy-free protocol will be summarized by high-level functions enumerated below. These functions can be considered at the logical level of the Xlib interface [8], and similarly, each one would correspond to a particular protocol request or event.[†]

2.3.1 Access Control Mechanisms (including cut and paste)

The proposed framework implies that the X Server assume the role of Access Control Enforcement Function (AEF) and that the PDC be the Access Control Decision Function (ADF). Basically this means that the X Server sends an event when a subject attempts to access an object, and that the PDC replies with a request indicating (1) whether the access was granted, and (2) if not, what error condition is represented by the access denial.

This separation of duty allows a clean implementation of *mediated* policies. The most well-known of this class of policies is the CMW cut and paste policy. In this policy, when a subject attempts to paste previously cut data into a target window whose security classification differs from that of the source window, an interactive window session is required before the paste operation can complete. This interaction reminds the user of the source and target window classifications and asks the user to verify the reclassification. Such an interaction can be handled cleanly and without race conditions if the PDC initiates the interactive verification session upon receiving the appropriate access control request. Other interesting mediation policies can be similarly implemented.

The separation of duty also allows a straightforward implementation of floating information labels as specified by the CMW requirements. In particular, the PDC can implement the policy whereby a successful "read" access to a particular object by a particular subject results in a change to the ACI of the subject, based on the relationship between the client's current ACI and the newly accessed object's ACI.

The following mechanisms are sufficient to implement access control:

- **RequestAccess(entityID, entityType, clientID, accessMethod)** event. The X Server sends this "event" (really a request) to the PDC whenever a subject (specified by **clientID** attempts to access an entity[‡] (specified by **enti-**

[†] A *request* is a protocol message from an X client to the X Server; an *event* or *response* is a protocol message from the X Server to an X client. The X protocol is asynchronous; the result of a failed request is typically an error response. Note that a protocol message is classified as a request or as an event based solely on whether it is input to or output from the X Server.

[‡] An **entity** is a subject or an object.

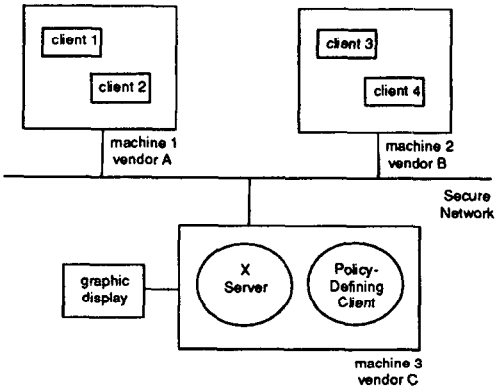

Figure 2: An X window topology with a policy-defining client

tyID). The X Server also sends the entity's type (**entityType**).

Entities in the X Server must be classified by type in order to take advantage of type-based policies. The decomposition of X Windows entities into types has been treated elsewhere (e.g., [2]) and is outside the scope of this paper. We generalize the usual object decomposition somewhat by allowing the definition of subject access policies. This is necessary in order to implement a generalized form of privilege described later.

- **AccessAnswer(yesno)** request. This "request" (really a reply) is the PDC's reply to the **RequestAccess()** policy question. **yesno** contains the answer, either 'yes' or an error return defining what the access denial policy is for this particular access request.

Note that the PDC need not send an immediate **AccessAnswer()** reply upon receipt of a **RequestAccess()**. Rather, as described above, it can implement a mediation policy on the access, or even delay the request to throttle a covert channel, before replying.

A **RequestAccess()** might also signal the PDC to create a new access control binding. For example, the successful attempt to create a new object would typically cause the PDC to synthesize new ACI (perhaps from the subject's ACI) to be bound to the object.

2.3.2 Access Control Information Binding Mechanisms

There must be a way for X to bind ACI to subjects and objects. In order for the protocol to remain policy-free, this ACI must be uninterpreted by the X Server. The protocol should therefore provide only a transport mechanism and binding semantics for ACI.

An important consideration is the location of the bindings between entities and ACI. These bindings should be stored in the PDC, since it must be able to make access decisions based on current bindings. If the X Server stored the bindings, it would have to verify the validity of every **AccessAnswer()** request by checking the existence of the accessed object.

The ACI binding may be rather static (e.g., Bell-LaPadula mandatory access) or may be rather dynamic (e.g., CMW subject floating information labels or object ACLs). The potential for dynamism requires that the binding mechanisms be as general as possible, allowing the PDC fine-grained control over the ACI.

The following mechanisms are sufficient to implement ACI binding policies:

- **BindClientACI(clientID, clientACI)** event. The X Server sends this event to the PDC. The event contains the handle of a client (a **clientID**) that just connected to the X Server, and the ACI of this client as reported *from the network*. Thus, a network supporting a secure X Window system must be able to provide this service; historically, similar services have been proposed for secure network services such as MAXSIX [13].[§] The PDC binds the **clientACI** to the client denoted by **clientID**.[¶]

[§]In general, any secure distributed system must include a networked identification and authentication service; otherwise a remote trusted server cannot enforce policy. A formal definition of this network service is not an X interoperability issue and is beyond the scope of this paper.

[¶]The format of the client ACI should be general enough to handle all types of access control information in a machine-independent fashion. The formal specification of this format should be defined by a standards body and will not be further explored here.

- **BindObjectACI(objectID, objectACI)** event. This event is similar to **BindClientACI()** and requires that the PDC bind the access control information to a particular X Server object. It is expected that this event is the result of a privileged client's desire to change the ACI of a particular object (see the "Policy emulation mechanisms" section below).

  The PDC is also capable of binding object ACI unprompted; for example, as a side effect of a successful **RequestAccess()** event requesting a new object be created. In that case, a reasonable policy would be that the newly created object is bound to ACI derived from the creating client's bound ACI.

2.3.3 Privilege Assertion Mechanisms

There are two general classes of privilege which are relevant to the protocol framework. First, there are subject privileges which are used by the PDC in order for it to implement windowing security policy. Many vendors desire to define fine-grained privilege policies which allow clients the right to enable and disable their own current privilege set (*privilege bracketing*: see [4]). The following mechanism is sufficient to allow *in-band* (that is, X protocol) privilege bracketing:

- **BindClientACI(clientACI)** request. This request has syntax similar to the **BindClientACI()** event described above. In this case, the client requests that its privilege set be changed as specified by the **clientACI**. The X Server forwards this request to the PDC using the **BindClientACI()** event. The PDC defines a policy which decides whether to honor the ACI (privilege) change request, presumably based on the ACI already bound to the requesting client.

Note that it is possible for the PDC to grant fine-grained policy-defining privileges to other clients as it sees fit using only the mechanisms supporting this first class of privilege.‖

‖One possible use of this mechanism would be to support a complex privilege-bracketed information labeling scheme. For example, a vendor may wish to implement a policy whereby the *xterm* terminal emulator will change the visible information label in a window based on the information in that window. One way to implement this policy is as follows: (1) *xterm* is granted the "create TCB-private window" and "change privilege to write to TCB-private window" privileges by the secure OS; (2) the *xterm* is bound to these privileges when it connects to the X Server; (3) *xterm* creates a TCB-private label window, which the PDC allows; (4) when it notices that it must write an information label, *xterm* requests that it be given the right to write into the TCB-private area and the PDC grants the privilege; (5) *xterm* writes the new information label into the TCB-private window, which the PDC allows; and (6) *xterm* relinquishes its right to write into the TCB-private window. It is worth noting that the X Server was never aware of the semantics of either of the privileges used in this scenario.

The second class of privilege is the class of policy-defining privileges, in particular, the privilege to be a policy-defining client. A mechanism must exist allowing the PDC to declare itself as a PDC to the X Server. The following mechanism is sufficient to support the policy-defining privilege:

- **AssertPrivilege()** request. This request simply informs the X Server that the requesting client wishes to be the PDC. The first client requesting the privilege is granted the privilege. If the request is not granted, the requesting client receives a failure notification.

It is required that the underlying secure operating system provide a trusted path and trusted startup semantics so that the PDC is guaranteed to be the first X client to send this request. (One possible implementation of this trusted path would be for the UNIX *login* program to start up the PDC, which would in turn start up the X Server and send it the **AssertPrivilege()** request. Should a spoofing PDC intercede, the trusted PDC would receive a failure notification and would be able to terminate the session and audit the spoof. Since the real PDC is using an underlying network trusted path, it can trust that the reply from the **AssertPrivilege()** is genuine.)

2.3.4 Policy Cognizance Mechanisms

There needs to exist mechanisms whereby a client can be *cognizant* of the policy that PDC defines. An example of such a client would be a *gadget manager*, where a *gadget* is an object defined by the client (and whose type is not known by the X Server). A client may define a gadget such that it emulates an X Window, for example. Such a client would need to know what window policy is being defined by the PDC in order for it to emulate the same policy for the objects it defines.

The mechanisms to support this cognizance capability are basically extensions of mechanisms already defined. These mechanisms are as follows:

- **RequestAccess(entityType, entityACI, clientACI, accessMethod)** request. This request has syntax similar to the **RequestAccess()** event described previously. In this case,

however, the policy cognizant client sends the request to the X Server, which forwards it to the PDC. The PDC then sends the reply back to the X Server as defined above. Note that the policy cognizant client must specify the type and ACI of the accessed entity, and the type of the accessing client, in order to find out what policy the PDC enforces.

- **AccessAnswer(yesno)** event. When the X Server receives the answer from the PDC, it forwards it back to the emulating client as an event.
- **GetACI(entityID)** request. The policy cognizant client requests that the ACI bound to **entityID** be returned to it. The X Server forwards the request to the PDC. It is expected that the PDC will require that the requesting client pass access control checks before it returns the bound ACI.
- **BoundACI(entityACI)** event. The **entityACI** is returned to the requesting client; or if the PDC disallows the request, null ACI is returned.
- **BindObjectACI(objectID, objectACI)** request. This request allows the policy cognizant client to request that the PDC bind new object ACI to the specified object. It is expected that the PDC will require that the policy cognizant client possess an appropriate privilege.

The X Server is required to mark the **AccessAnswer()** and **BoundACI()** events with a tag indicating that the answer is genuine; otherwise a malicious client could use the generic X Window event mechanisms to spoof the PDC.

These mechanisms provide a simple method for a client to find out what policy the PDC is defining. For simple policies (e.g., strict MAC with a few labels), this method is sufficient; for more complex policies, it may be necessary for the policy defining client to make assumptions about the PDC policy. The problem is analogous to the problem of a client that wishes to be cognizant of the window management policy; in that case, the X Protocol also provides only basic information about the policy and a client needing to know more would have to make assumptions based on the documentation describing the window manager.

2.4 Backward Compatibility

It is important for an X Server implementing the above mechanisms to maintain a *backward compatibility mode* so that customers can choose to enable or disable the security policy as desired. The backward compatibility mode is simple in this case: if no client declares itself as a PDC, the X Server will not issue any **RequestAccess()** events, and the server will implement its original policy. Also, should a policy cognizant client issue a **RequestAccess()** or **GetACI()** request, the X Server will always return an **AccessAnswer()** *yes* or **BoundACI()** *null*, respectively. This retains interoperability in backward compatible mode.

2.5 Security through Encapsulation or Separation

Several vendors have attempted to implement security policy by *encapsulation* or *separation*, whereby the X Server runs untrusted. In an *encapsulation* architecture, there is typically a small, trusted X Server emulator which handles a limited set of trusted windowing operations [4]. Alternatively, for the *separation* architecture, a secure network and trusted X clients could be configured to implement secure windowing policy, without the need to implement a multilevel X Server.

These architectures can be made interoperable by defining a simple PDC that emulates the original X Window policy by giving a "yes" answer to any policy questions. (A PDC must be defined; otherwise, another client could spoof the PDC simply by doing an **AssertPrivilege()** request.) In general, encapsulation or separation architectures are defining virtual machines, where individual clients (even security-cognizant ones) *should not* be aware of the underlying window security policy.

2.6 Performance

The proposed protocol framework has performance implications. In particular, nearly every X Window request will cause the generation of one or more **RequestAccess()/AccessAnswer()** transactions. This potential performance problem can be solved or mitigated in several ways.

The first way is by taking advantage of local configurations. Typically, a machine supporting the X Window System also supports an in-memory local client connection facility, whereby clients running on the same machine as the X Server communicate via shared memory. If the X Server and the PDC are on the same machine, the **RequestAccess()** overhead should be considerably lessened.

The second way, which is an extension of the first, is for X terminals to support the proposed protocol.**

**The possibility of this happening is largely predicated upon

Typically, X terminals provide enough memory for some clients to reside in the terminal firmware along with the X Server. X terminals also often provide downloading capabilities. Such capabilities could be used to create an X terminal-local configuration similar to the first method above. In this case, the performance should be even better, because the X terminal is dedicated to X Window operations.

The third way is for the X Server to provide an access decision cacheing facility, where the PDC's prior decisions are remembered by the server for later decisions. It is expected that many **RequestAccess()**s will be identical (or at least that the relevant ACI will be identical for many **RequestAccess()**s), so there would be a high cache hit ratio over the lifetime of an X Window invocation.

While it is possible to define a reasonable cacheing scheme to take advantage of these properties, it is not yet clear if it is really necessary. Such a scheme would complicate the protocol[††], and in order for the scheme to be interoperable, This complication would have to include, among other things, a cache flushing mechanism to allow the implementation of *time-based* policies such as RELEASEABLE AT ¡time¿–see [3]. vendors would have to agree on a particular cacheing protocol, perhaps before the problem is completely understood. For these reasons, we have chosen not to include a cacheing scheme in the proposed framework.

The fourth way is to implement the PDC not as a separate client but as a dynamically loadable library linked to the X Server. This method has been prototyped by the author, using library procedure calls in place of protocol transactions between the X Server and PDC, as a basic proof of concept of the protocol interface and as a simple performance modeling mechanism. The prototype implemented a very simple DAC policy. It did *not* contain the policy cognizance mechanisms. The prototype confirmed that many **RequestAccess()/AccessAnswer()** transactions occur in the startup phase of the X clients from the standard MIT distribution. However, after this small initial delay, no other delays based on this simple policy were noticeable. More performance modeling must be done with more complex policies before the mechanisms can be deemed practical.

the protocol being standardized by the MIT X Consortium as described earlier.

[††]This complication would have to include, among other things, a cache flushing mechanism to allow the implementation of *time-based* policies such as RELEASEABLE AT <time>—see [3].

3 Implications for X Server and Policy Defining Client

The proposed framework makes certain assumptions about the behavior of the X Server and the PDC.

First, the framework does not indicate what, if any, steps should be taken by the X Server or the PDC to alleviate denial of service attacks. For example, the framework does not dictate that only trusted clients be able to use the **XGrabServer()** request; **XGrabServer()** tells the server to listen only to the client issuing the request until further notice. There are many other ways that a malicious client could degrade service through normal X Window requests. For the proposed framework, the X Server should be able to translate at least some of these denial of service problems into access requests that the PDC can act upon. For the example above, one reasonable solution would be for the X Server to define a SERVER object type and to issue a **RequestAccess()** requesting WRITE access to that SERVER object for the requesting client. The PDC can then decide if it should restrict access to this particular operation.

To generalize somewhat, the proposed framework assumes that the decomposition of the X Server into objects, object types, and access methods be done in such a way that *all reasonable policies* can be implemented. It is not clear by which criterion one should classify policies as reasonable; however, experience with existing secure X Window system models should be very helpful in this regard. One possible difficulty here is the creation of an X Server that is cognizant of the relationship between requests and events so that covert channels can be treated as access requests.[‡‡]

A further assumption is that the infrastructure (the secure OS and the secure network) provide facilities that do not compromise the security policies defined by the PDC. For example, there must be a trusted path to the PDC so that another client cannot spoof it. Also, the X Server must not allow access to an object when that object's ACI is being bound (the *tranquility* property). Finally, there must be a trusted path between the X Server and the physical display to preserve the integrity of any security relevant output (e.g., visible labels) or security relevant input (e.g., a security marking created at the user's discretion).

[‡‡]Epstein [4] notes that the most difficult of the covert channels is the window exposure problem whereby one client can signal another client through the exposure of a previously covered window. The X Server must have a **RequestAccess()** strategically placed so that the PDC can determine the ACI of the exposing client and of the exposed window, and make a decision based on their relationship.

It is important to note that the implementation of the proposed framework alone is not sufficient for the X Window System to be certifiable past B1 or B1/CMW. A modular restructuring and covert channel analysis of the X Server, or possibly the implementation of an *encapsulation* or *separation* trusted X architecture as previously described, would also be necessary preconditions for B2 or B3 certifiability.

4 Implications for X Window System Embedded Base

A major advantage of a policy-free interface to the X Server is that the vendor can decide what impact the security policy will have on the embedded COTS client base. For example, the vendor may choose to implement a restricted form of the ss-property by making all objects invisible to a client unless their MAC labels are equal. Such a policy would tend to be useful in a system where the customer site is interested in strict separation of labeled data; however, such a policy has an impact on administrative COTS clients such as *xlswins* that is different from the impact of a "read down" policy.

Often, the vendor will be choosing between defining a policy that reports that an access failure is due to the nonexistence of an object, and a policy that reports that the failure is due to a security violation. The vendor also has the opportunity to construct clever subpolicies such as (for example) defining certain objects as public, in order to provide greater compatibility with a particular COTS client.

The vendor can also make use of sophisticated policies in the attempt to provide compatibility. For example, a vendor willing to analyze the behavior of a particular COTS client might write a PDC defining a particular privilege that allows the client to access data that the invoking subject could not.

5 Implications for Other Trusted Systems

It has been noted [2] that the historical absence of a security policy has hampered the effort of reaching a consensus on a secure X Window system, largely because vendors have tailored the X protocol to provide the level of security and compatibility that they thought necessary. However, it is also true that the absence of a standard protocol has allowed vendors to explore many implementation possibilities, and in so doing there are now a set of *de facto* requirements for the support of various policies and implementations in any standard. From this point of view, the X Window system is in a superior position relative to other systems with premature *de facto* standard policy interfaces. Probably the best example of a premature policy standard is the UNIX discretionary access control policy and implementation, which (1) cannot be changed, and (2) cannot be described in fewer than ten complex rules(!).

It is much easier to *add* (re-engineer) a new policy interface than it is to *change* (reverse engineer) an existing one. The reverse engineering problem is that the goals of security (requiring a clear formulation) and compatibility (requiring no change to an old, unclear formulation) are at odds. This in turn implies that a rearchitecture of an existing system such as UNIX along the lines of the proposed framework would be problematic at best. The problem has been faced by the ORGCON prototype project [3] (among others); in that case, architectural purity was sacrificed for expediency.

For newer trusted systems, the methodology implied by the proposed framework has general applicability. Specifically, the trusted system designer is forced to face the following question: Given a target policy problem space, what is the simplest and best-performing mechanism that can be built that will support the entire policy problem space? It has often happened that the general applicability of a trusted system is not realized until it is fielded and new security requirements are generated based on field experience. The separation and decomposition methods described above would be a hedge against this eventuality.

6 Conclusions

By abstracting the security policy decision-making function away from the policy enforcement function, a simple, mechanistic interface will often become apparent. Such an interface has the potential of being both *non-controversial* and *extensible* to a large class of security policies.

We examined the effects of constructing such an interface for the X Window System, whose very reason for existence is to support a large, distributed, heterogeneous, open, and evolving graphical user interface environment. The approach appears to be most promising in systems with those qualities; the applicability of the approach is less evident in systems that are relatively small, monolithic, proprietary, or unchanging over time.

References

[1] Grenier, G., R. C. Holt, and M. Funkenhauser, Policy vs Mechanism in the Secure Tunis Operating System, *IEEE*, p. 84, 1989.

[2] Faden, G., Reconciling CMW Requirements with Those of X11 Applications, *Proceedings of the 14th National Computer Security Conference*, Washington, D.C., October 1-4, 1991.

[3] Abrams, M. et al, Generalized Framework for Access Control: Towards Prototyping the ORGCON Policy, *Proceedings of the 14th National Computer Security Conference*, Washington, D.C., October 1-4, 1991.

[4] Epstein, J. and J. Picciotto, Trusting X: Issues in Building Trusted X Window Systems or What's not Trusted About X?, *Proceedings of the 14th National Computer Security Conference*, Washington, D.C., October 1-4, 1991.

[5] Rosenthal, D., *LINX–a Less INsecure X server*, Sun Microsystems, April 1989.

[6] Picciotto, J., *Trusted X Window System*, MTP 288, The MITRE Corporation, February 1990.

[7] Scheifler, R. and J. Gettys, *X Window System*, Digital Press, 1990.

[8] Gettys, J., R. Scheifler, and R. Newman, *Xlib–The C Language X Interface*, Silicon Press, 1989.

[9] Graubart, R., J. Berger, and J. Woodward, Compartmented Mode Workstation Evaluation Criteria, Version 1 (Final), DIA Directorate for Information Services, 1991.

[10] O'Brien, R. and C. Rogers, Developing Applications on LOCK, *Proceedings of the 14th National Computer Security Conference*, Washington, D.C., October 1-4, 1991.

[11] Carson, M. et al., Secure Window Systems for UNIX, *Proceedings of the 1989 Winter USENIX Technical Conference*, San Diego, CA, Jan 30-Feb 3, 1989.

[12] *Access Control Framework*, CD10181-3, ISO/IEC JTC 1/SC 21 N6188, June 24, 1991.

[13] Department of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985.

[14] *DNSIX 3.0 Architectural Overview, Rev 1*, SecureWare, April 1992.