Securing The X Window System With SELinux

Doug Kilpatrick
NAI Labs

dkilpatr@nai.com

Wayne Salamon
NAI Labs

wsalamon@nai.com

Chris Vance
NAI Labs

cvance@nai.com

This work supported by NSA contract MDA904-01-C-0926 (SELinux)
Initial: January 2003, Last revised: March 2003
NAI Labs Report #03-006

Table of Contents

1. Introduction

The X Window System, or 'X11', has become the standard graphical engine for the Unix and Linux operating systems. Its network-based design and platform independent support for a wide range of operating systems and hardware has contributed greatly to its acceptance. [XOrgIntro]

The X protocol was designed with compatibility and performance in mind, not security. However, since the X protocol is a constrained channel of communication, it enables the enforcement of a security policy While there has been quite a bit of research done in the past to secure X11, many solutions are specific to the government's Multi-Level Security (MLS) model, and are not in widespread use.

NSA Security-Enhanced Linux (SELinux) [SELinux]. is an implementation of Flask, a flexible and fine-grained mandatory access control (MAC) architecture [FlaskArch]. SELinux can enforce an administratively defined security policy over all processes and objects in the system, basing decisions on labels containing a variety of security-relevant information. The architecture provides flexibility by cleanly separating the policy decision-making logic from the policy enforcement logic. The policy decision-making logic is encapsulated within a single component, known as the security server, with a general security interface. A wide range of security models can be implemented as security servers without requiring any changes to any other component of the system. The design and implementation of the SELinux prototype is described in [LoscoccoFreenix2001] and [LoscoccoNSATR2001], both of which can be found at the NSA SELinux web site (http://www.nsa.gov/selinux).

On a current SELinux system, applications can use the X server as an additional communications vector, unregulated by the system policy. In addition, applications can manipulate the X server to attack other client applications, or to mislead the user. By running the X Server on an SELinux system, and by extending the FLASK architecture to allow the X Server to act as a trusted application, the security of the user operating environment should be enhanced.

This paper assumes familiarity with the Flask architecture and its Linux implementation. The paper starts with an overview of the X11 architecture and desirable security functionality. Section 5 lists the object classes that will need to be labeled, the permissions those object classes support, and the control requirements for each of the X11 protocol operations. The security features for error and event processing are then described in Section 6.5 and Section 5.3. Finally, Section 7 discusses security-aware applications.

2. Overview of the X Architecture

The X Window System is a windowing system for bitmapped graphics displays. It is based on a client-server architecture. The server controls the display and associated input devices, the clients are the graphical programs that access those services.

Clients connect to the server via Unix domain sockets (if local) or TCP/IP (if remote). (Other transport mechanisms can be supported, but are less common) Clients pass requests to the X server and receive events from the server by a clearly defined protocol [OReilly90], the XProtocol. All communication between the X server and the clients, with the exception of some image sharing extensions, happens over this connection.

The clients can only communicate with the X server through this connection, and can not directly communicate with other clients over this connection. However, the actions a client takes will frequently be visible to other clients. Some desktop environments (e.g. GNOME) expect applications to communicate with each other via other forms of IPC. These communication vectors are outside the scope of this paper.

Figure 1. X client communication

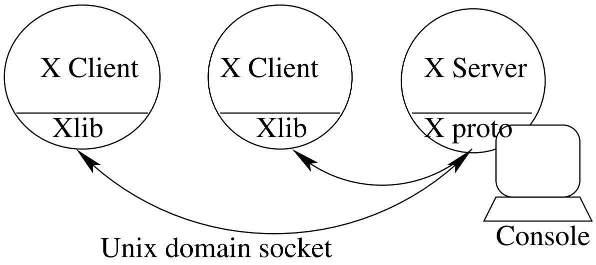


Figure 1 shows two clients connected to a local X server. The client code usually uses the Xlib library to handle marshaling requests with the X protocol, but this is not required.

The X server also maintains certain data structures as resources that can be shared between clients. Clients refer to these resources by resource IDs passed over the communication link. Client applications frequently use these resources to communicate with other client applications, effectively using the X server as a communications channel.

3. Threats and Security Concerns

In order to design a set of protections needed within the X server, the set of vulnerabilities that are present in the current X server were investigated. The threats can be broadly classified into the traditional security property categories of confidentiality, integrity, and availability. The protections presented in this paper are designed to prevent compromises of these properties by X clients. We also discuss other relevant properties of secure systems, such as covert channels and the assurance of the system.

The X server does not exist as the only secured element on an otherwise untrusted system. Rather, we assume the X server is running on a secured SELinux system. The kernel enforcement layers will prevent unauthorized clients from connecting to the server or otherwise affecting the kernel structures of the server. Our design only considers clients that successfully connect using the base X Protocol.

We did not investigate attacks directly on the X Server. For example, a buffer overflow may be discovered which allows a client application to control the X server. Such attacks are outside the scope of this work. This analysis is only concerned with syntactically valid series of X11 requests.

3.1. Confidentiality

There are many ways applications could use a standard X server to bypass a confidentiality policy. As screenshot applications demonstrate, standard X servers offer no protection for an application's output. Clipboard managers can grab content from an application automatically, without any malicious intent.

Malicious applications can currently probe the entire state of the X server. By manipulating the input to a client process, a malicious application could cause the victim process to violate a confidentiality policy. The protections presented in this paper will make it possible for a policy to protect the confidentiality of data presented by X clients.

3.2. Integrity

The current X server offers little protection for the integrity of client program data. The X server does not interact directly with the data of the client. Clients can draw directly into another client's window and can send input, or any other type of event, directly to a client. A malicious client can cause another client to present false information to the user. Or an attacker could insert malicious commands into the input stream of a terminal emulator. These types of attacks can cause an application to violate the system's integrity policy, and the protections presented in this paper are designed to protect the integrity of X clients.

3.3. Availability

The standard X server offers little protection for the availability of the server. Clients can close the windows of other clients, manipulate the font lists, and manipulate the host access lists. The design presented here provides minor improvements for the availability of the X server. While a client will still be able to cause the X server to use excessive CPU via various denial-of-service attacks, the protections presented in this paper will allow a policy to protect the X server against other availability attacks.

3.4. Assurance

The focus of this work is to develop a security framework for the X server which can support strong, flexible security policies. While system assurance is an important element for a secure system, assurance is beyond the scope of this paper. This is commensurate with the assurance effort associated with the underlying system, SELinux, that "has not been on system assurance or other security features such as security auditing, although these elements are also important for a secure system. " [SELinux] The primary goal is to develop a MAC framework for the X server. We expect that formal means of verifying the design should be used.

The X server is a large system component, with a large code base that is hard to evaluate for correctness. Other mechanisms for providing separation between security domains (e.g., various poly-instantiation techniques) would provide increased assurance, but would not be as adaptable to the large number of domains in an SELinux system.

The X server is extensible and still under development. Addressing the common extensions in use and those being developed is beyond the scope of this paper. However, in order to provide complete coverage for the X server, the common extensions will also need to be evaluated from an enforcement perspective.

3.5. Covert Channels

The SELinux project has not, in general, addressed covert channels. "Even with covert channels, an operating system with basic mandatory controls improves security by increasing the required sophistication of the adversary. Once systems with basic mandatory controls become mainstream, covert channel exploitation will become more common and public awareness of the need to address covert channels in computing systems will increase.” [LoscoccoNISS1998]

As with SELinux, the protections presented here do not consider covert channels. However, this design should be compatible with attempts to manage covert channels. A more comprehensive design which considers covert channels will need enhanced visibility controls, and may also need to limit the visibility of changes to the global server objects.

4. Requirements

The X protocol for graphical applications was not designed with security as a major concern. The X server applies limits to initial connections from clients, but does not limit actions of connected clients. Any client on the system can read the state of any objects in the server, can receive and monitor any and all input, and can manipulate any window.

The base protocol describes 120 separate operations on at least 15 different types of objects. Until the creation of the X11 Security Extension [WigginsExtension96], no privilege separation was provided. Even now, the X11 Security Extension simply separates applications into two domains, one of which is protected from the other. The Security Extension does not provide fine-grained access control. Fundamental actions in the protocol, such as cut-and-paste, assume permission to modify objects belonging to other clients, limiting the ability of compatible systems to perform extensive access control.

Normal use of an X window system requires a window manager, an application trusted to manage the windows of all other clients. By necessity, the window manager needs permission to move, resize, and hide windows created by other applications, giving it permission to subvert nearly any existing security policy.

4.1. Required Functionality

One approach for preventing inappropriate communiation between clients of different security domains is to assign each security domain its own X server, removing all communication channels via the X protocol. For a user to run two applications in different security domains, the user would need to run two different X servers. This approach would make it harder to use arbitrary X applications, and might force the user to be continuously restarting the X server.

If the server is allowed to interact with applications in different domains, but does not enforce any policy on the applications, the running applications can use the X server as an unregulated means of communication. Malicious applications can subvert the system’s security policy and gain control over other programs in other domains by exploiting the X communication protocol.

A secure X server needs to operates within the confines of the system security policy, while enabling the users to perform theirs tasks. To achieve this goal, the X server needs to enforce the X11-related aspects of the system policy, and understand the relationship of its clients under that policy. The server needs to continue to communicate with clients in different domains, yet prevent those clients from communicating through the server, and it needs to protect the user from malicious clients.

4.2. Requirements for Securing X

Vendors have previously adapted the X protocol for Multi-Level Security (MLS) systems. The NSA and DoD specified the requirements in the Compartmentalized Mode Workstation (CMW) specifications. These specifications are primarily concerned with performing access control on the server objects according to the MLS policy, but they also require a few extra security features such as window labeling and trusted input paths.

While a secure X server for an SELinux system does not need to fulfill the same requirements as an CMW, these requirements serve as a useful starting point and basis for comparison.

4.2.1. Labeling

In CMW systems, top-level windows need to be clearly marked with their current security domain. The user must be aware that the window he or she is using is considered secret or unclassified. The window system, along with the window manager, is expected to clearly mark windows with visual labels to pass this information along to the user. Applications must be prevented from spoofing the label and misleading the user. In addition to visual labels that indicate which windows are in which security contexts, visual labels that clearly indicate which windows are capable of receiving input are required.

The visual labeling requirements for a CMW may not apply to SELinux systems running the default policy. The SELinux Security Identifier (SID) can encapsulate more information than just the classification level of a process. In an SELinux system, applications are expected to operate in many different domains and unique, clearly distinguishable visual labels for each domain may not be practical.

While individual windows in an X application rarely contain child windows from another client, it is possible. For example, the preview screen for screen savers in both KDE an GNOME execute the screen saver in a small window as the preview pane. This possibility requires us to track data flow on all the window objects in the X server, and apply policy to sub windows as well as parent windows.

If a secured X Window system needs to provide a visual labels, this functionality will be provided by the window manager, which is considered to be a trusted client. The window manager is responsible for determining input focus, so adding CMW-style labeling is an appropriate function for the window manager. The secured X server will provide information to the window manager to enable it to add the visual labels, and will protect the window manager from malicious clients.

4.2.2. Trusted Path

Trusted path is a mechanism by which a user can communicate directly with the trusted computing base, providing confidentiality, integrity, and availability on the input channel. Trusted paths are used for entering particularly sensitive information, like authentication tokens.

The trusted-path requirements for graphical displays are linked to window labeling, as they require that the user be made aware of where their input is and require that their input not go elsewhere. Ensuring a trusted path is most critical at login, so key sequences guaranteed to reset the system to a login state may be sufficient.

Currently, X provides very little protection for an application's input stream (keyboard input, mouse actions). Any client can grab the input stream, observe, discard or otherwise modify it, or fabricate input before passing it to what appears to be the active client application. Any client can open a screen-covering input-only window, catching any otherwise non-grabbed input. The protections developed need to allow a policy to prevent arbitrary programs from grabbing the input and from forwarding faked events, but they do not need to prevent clients from snooping non-grabbed input. To prevent classic X input snooping, the window manager must be trusted. The window manager can then relabel the input stream according to its knowledge of input focus. The secure X server can then apply permissions to input events, allowing the policy to limit their visibility.

4.2.3. Window Data Snooping

X clients can easily retrieve the bitmaps of other client's windows, as demonstrated by the availability of screen shot applications. This ability allows client programs to violate a systems confidentiality policy. The Security Extension already in common X servers provides some protection by preventing untrusted applications from accessing the window data of trusted applications, but it specifies a rather limited policy and is not widely used.

There are two approaches clients can use to get screen shot information. The first is to simply get the information directly from the window via the `GetImage` request. The secure X server must allow the policy to prohibit this action. The second approach is to create a new empty no-background window over the screen, and then look at its contents. Optimally, the secure X server could enforce a policy that prohibits windows that have no background. The policy we present here does not have sufficient granularity to express this restriction, as window attributes are not suitable objects classes for SELinux. The policy enforcement layer must recognize that mapping a window may involve copying the contents of a window being covered. The server must check for that permission and force some kind of default background upon the window if the permission to copy is denied.

4.2.4. Cut-and-Paste

The cut-and-paste protocol relies on the ability of an application to set properties on the window offering the data [XSelections]. Unfortunately, properties are used for everything from drag-and-drop to instructions to the window manager necessary for the proper operation of the application. Properties are name/value pairs defined outside of the X protocol with specific use dictated both by convention and other standards documents (Internet RFCs). Clients must cooperate with each other and the X server to perform cut-and-paste operations.

A policy can enforce confidentiality and integrity requirements by preventing a client from setting window properties on windows belonging to another client. This policy will also prevent cut-and-paste from working between these clients. If cut-and-paste is required functionality, a mechanism must be developed to mediate cut-and-paste actions. Previous approaches have created a trusted cut-and-paste server that manages all requests, limiting the communication between the two processes.

Another approach would require that the properties of a window be labeled. This would require the use of a property object as a separately managed object inside the X server. This object would be suitable for labeling despite not appearing as an object type in the X protocol. However, without a separate labeling-by-name step, there is no way to separately label the cut-and-paste relevant properties from other client-specific properties. In fact, the property used for passing the actual data in a cut and paste operation is arbitrarily chosen by the client requesting the data.

A trusted window manager could provide a solution, but only in combination with the proposed clipboard related extensions currently under development by XFree86 [ClipExt]. One proposed extension would allow clients to register for notification when the active selection changed. The window manager will claim the selection when that happens and place it in a window with a special label. The policy will allow or deny property reads and sending events to that window.

4.2.5. Application Compatibility

Well-behaved X applications must continue to run unmodified on a secure X system. This high level of transparency would give an SELinux system a large catalog of useful graphical applications and ease development of other applications. Certain applications (e.g. screen-shot programs) may not run, but that is a desired result of policy enforcement.

If possible, actions that would violate the system's security policy should fail cleanly. For example, a paste operation of high confidentiality data into a low-confidentiality client should simply fail, as if no data were available in the clipboard. Screen shot applications should simply fail to accurately represent the state of the screen. Clients should not, in general, see X protocol errors which might cause them to abort abnormally.

4.2.6. Scope of Changes

The changes to the XFree86-based X server need to be minimal as XFree86 is an evolving system. Small, self-contained changes are easier to adapt to changing systems than large intrusive changes, tend to be easier to understand, and are more likely to be accepted by project maintainers.

Any changes to the X system should be suitable for inclusion into the XFree86 project so that SELinux systems can gain greater use. Prior experience in proposing changes to software maintained by others has shown that large complex patches are hard to review are often not applied. If the changes can be pushed into the primary XFree86 tree, they are more likely to be picked up by other systems, and will ease the support burden of the SELinux project.

4.2.7. Placement of Trust

The expectation, presented below, is that the X server will operate as a trusted application, capable of enforcing security decisions. We assume that the X server is able to enforce a wide range of policies, and able to communicate simultaneously with clients in separate domains.

The SELinux module enforces all security decisions in the kernel, protecting the enforcement mechanism from tampering by malicious agents. A system policy will be created that will prevent users from directly controlling the X server. If further assurance is required, then various poly-instantiation approaches could be used[Kirk97], where the system would limit which clients could connect to any given X server, but many X servers could be running at any one time. Particularly in the SELinux model of many domains, a poly-instantiation based approach may consume excessive resources to be practical, but is possible within the design we present.

5. Security Architecture for the X Protocol

The X protocol specifies four types of messages that can be transferred over the network. Requests are sent from the client to the server, and replies, events and errors are sent from the server to the client. Replies contain data requested by a client. Events represent asynchronous activity the client has expressed interest in. Errors are normally sent in response to malformed requests, but can occur at any time.

Requests manipulate server side objects, called resources, referenced by an integer resource ID. Resources are created by requests, and are freed by requests or when connections are closed. Most resources are potentially sharable between applications. Resources are identified in the X protocol by integers. The client that creates the resource gives an initial identifier for it. The X server may change that identifier, usually adding a client id to the high bits of the integer. Clients normally get resource identifiers of objects in the X server by using various query methods. However, clients can reference unknown resources by guessing the resource identifier.

Security labels are placed on objects that may be shared, and hence constitute a communication or control channel through which the system's security policy might be violated. Windows are manipulated explicitly by the client application and the window manager application. Fonts and cursors are shared automatically across multiple screens, and colormaps are frequently shared by applications.

Graphic Contexts (GC) are never shared between clients, and are not labeled and have no control requirements. Similarly, Fontable object are not labeled. Any request that takes a Fontable object will check the security information of the font passed in, or the default font of the graphic context.

The X protocol defines several other primitive types, such as integers of various sizes, time stamps, enum constants, and atoms. With the exception of atoms, these types in the X protocol do not represent objects in the server, and so can not be labeled. An atom is an integer type that references a string constant. As such, atoms have no significant role as objects in the server and are not labeled.

5.1. Design

This section describes the design for integrating the Flask security mechanisms into the X Window system. It begins with a discussion of the object classes and permissions defined for the X Window system component. This is followed by a description of the control requirements for the X protocol operations used to manage X server objects. Finally, an extension for security-aware applications is described.

5.1.1. Object Classes

The logical abstractions provided by the X Window system were studied to determine the set of object classes that need to be labeled and controlled by the security enhanced X server. The set of object classes for the X Window system component is shown in Table 1.

The Drawable object class does not represent a type of object actually found in the X server, although it is a type recognized by the X protocol. X requests that draw to an object currently take either a Pixmap or a Window as a target, and the Drawable will represent this object. For the purposes of defining permissions, the Drawable object class is defined to be a separate entity. In the modified X server, whenever a Window or a Pixmap object is created, a Drawable permission class will be used for permission checks related to draw operations.

Table 1. Object classes for the X Window system component

Object Class	Description
Drawable	A virtual object for output operations
Window	The base input and output object for an application
Font	An array of glyphs

Object Class	Description
Colormap	a color index
Cursor	cursor graphics
Client	An client to the X server

Server objects other than those listed in Table 1 will be labeled in the secure server. In particular, there will be labels for a single Server object and the Input object. Those objects do not have particular permissions, and so are not listed as object classes here.

5.1.2. Permissions

For each object class, a set of permissions is defined to control access to objects in that class. These permissions were identified by studying the services provided by the X Window system component. For each service, the objects whose state is observed or modified by the service were identified, and permissions for the corresponding object classes were defined.

5.1.2.1. Client Permissions

While client objects are frequently used as subjects to permission checks, the Client object class has only one permission, the permission needed to disconnect a client from the server. For most other checks, the context of a Window object, or other resource, will be used for the permission checks.

Table 2. Permissions for the Client object class

Permission(s)	Description
kill	Close down a client

5.1.2.2. Drawable Permissions

Table 3 shows the permissions defined for controlling access to the Drawable object class. Window objects and Pixmap objects are the actual arguments passed to drawing requests. The context of a Drawable object is derived from the window or client that created it.

Table 3. Permissions for the Drawable object class

Permission(s)	Description
create	Create a Drawable object
destroy	Destroy a Drawable object
draw	Draw into a Drawable object
copy	Copy pixels from a Drawable object
getattr	Get attributes from a Drawable object

5.1.2.3. Window Permissions

Table 4 shows the permissions defined for controlling access to the Window object class. A window is a rectangular area on the screen into which a client can draw. Windows are frequently nested and overlap. For example, a top-level window for Mozilla could have 20 or 30 windows inside of it. When a top-level window is created, it is assigned a context derived from the client that created it. When a child window is created, it is assigned a context derived from its parent.

Since windows are considered Drawable objects, and passed into drawing functions, the `create` permission of the Drawable is checked when a Window is created if the window is defined as an input/output window. The context of a new Drawable is derived from the context of the associated window, and thus from the context of the client. The context of the Drawable will be used for the checks for drawing functions.

A window also has a collection of properties. Properties are named bits of data, usually in the form of text strings. Example properties include window manager hints and cut-and-paste messages. Since the name is arbitrary, both the name of the property and the value of the property could be used to transmit information between clients.

Certain common property names are used by convention and are frequently set and queried. It would be useful for a policy to be able to differentiate between properties by name, but, without any initial labeling step, it may not be possible. It is possible to avoid labeling property objects, instead relying on a less fine-grained permission to change any window property. While there is sufficient state contained in a property object to justify labeling it, there does not appear to be any way to meaningfully distinguish between properties.

Table 4. Permissions for the Window object class

Permission(s)	Description
addchild	Create a new window
create	Create a new window
destroy	Destroy a window
map	Map a window
unmap	Unmap a window
chstack	Change stacking order
chproplist	Change property lists
chprop	Change a property
listprop	Query properties
getattr	Get window attributes
setattr	Set window attributes
move	Move window
chselection	Change selection
chparent	Change parent
ctrllife	Control lifetime
enumerate	List child windows

Permission(s)	Description
clientcomevent inputevent drawevent windowchangeevent windowchangerequest serverchangeevent	Send an Event

5.1.2.4. Font Permissions

Table 5 shows the permissions defined for controlling access to the Font object class. See Table 15 for the control requirements.

Increasingly, fewer applications use X server provided fonts objects. The anti-aliased font support from the Xft library actually loads the font metrics on the client, then uses the RENDER extension to draw them with transparency support. Mediation on font objects may not be needed except for legacy applications.

Table 5. Permissions for the Font object class

Permission(s)	Description
load	Load a font
free	Free (dereference) a font
getattr	Obtain font names, path, etc.
use	Use a font for drawing

5.1.2.5. Color and Colormap Permissions

Table 6 shows the permissions defined for controlling access to the Color object class. As with colormaps (below), it may be possible to assume static, high color displays, eliminating the need to track Color objects.

Table 6. Permissions for the Color object class

Permission(s)	Description
create	Create a new Color
free	Free a Color
lookup	Look up string name of color

Table 7 shows the permissions defined for controlling access to the Colormap object class. A colormap is a color lookup table. It is simply an array of colors in any order. The color of a given pixel is obtained by combining the pixel values with the colormap.

Colormaps are associated with windows. Before the colors of a given colormap are reflected on the screen, the colormap must be installed.

Colormaps only have meaning in relation to Visuals. Visuals can be grey-scale or color, static or dynamic. The most frequently used Visuals under Linux are `PseudoColor` and `TrueColor`. On a `PseudoColor` screen, pixel values index into a changeable colormap of red, green, and blue values. `PseudoColor` is usually used for 8-bit-per-plane color displays. On a `TrueColor` screen, pixel values are decomposed into separate red, green, and blue fields. Each component is then used as an index into a pre-defined fixed color lookup table. `TrueColor` is usually used for 24-bit-per-plane color displays.

Applications can share colormaps. Windows inherit the colormap of their parent, and most applications inherit the colormap of the root window. The standard dynamic colormap starts with just black and white defined, with other colors being defined as applications need them. An application that wants more control over the colormap can create its own colormap and then alter it.

A security policy needs to control colormaps both to prevent unwanted communication between client applications and to protect client applications from modifying the display of other applications. For example, a malicious application can blank out a terminal emulator window by setting the foreground and background colors in that window's colormap to be the same.

In recent years, the trend has been toward 24-bit-per-plane and higher displays, usually running the `TrueColor` visual. On displays with only static colormaps, the Color objects in the Colormap can not be changed by client applications, thereby reducing mischief. Control requirements for the full set of colormap definitions are provided in Section 5.2.4.

Table 7. Permissions for the Colormap object class

Permission(s)	Description
create	Create a new Colormap
free	Free a Colormap
install	Copy a virtual colormap into the display hardware
uninstall	Remove a virtual colormap from the display hardware
list	List installed colormaps
read	Read color cells of colormap
store	Change color cells in colormap
getattr	Get the color gamut of a screen
setattr	Set aspects of color conversion

5.1.2.6. Cursor Permissions

Table 8 shows the permissions defined for controlling access to the Cursor object class. Cursor objects are constructed with the `CreateCursor` request from a bitmap object, or more commonly created with `CreateGlyphCursor` from a font glyph. Normally the cursor is created from a standard cursor font. Cursor objects, particularly ones created from the standard cursor font, are logically local to a single client, but in practice are shared for the whole server. Cursor objects are associated with windows, and set via the `CreateWindow` request.

Table 8. Permissions for the Cursor object class

Permission(s)	Description

Permission(s)	Description
create	Create an arbitrary cursor object
createglyph	Create a cursor object from a font
assign	Associate a cursor object with a window
setattr	Set attributes of the cursor

5.1.2.7. Input and Server Permissions

Table 9 shows the permissions defined for controlling access to the Input object class. The `mousemotion` permission is listed here for detailed mouse motion events. However, there are no explicit permissions for normal mouse and keyboard input, as nearly all applications receive these events. If an application should not receive any input events, the policy can restrict the applications access to input events by controlling the `inputevent` permission in the Window Object class.

The permission to relabel the input device will be used by a trusted window manager to help properly direct input. See Table 19 for the relevant control requirements.

Table 9. Permissions for the Input object classes

Permission(s)	Description
getattr	Get input device attributes, such as keyboard mapping, pointer controls, etc.
setattr	Set input device attributes
grab	Grab server input, mouse or keyboard
passivegrab	Arrange to Grab server input, mouse or keyboard
bell	Ring the bell
mousemotion	Get Mouse Motion Events
relabelinput	Relabel the Input Stream

Table 10 shows the permissions defined for controlling access to the Server object class. Few applications need these permissions, generally only system configuration tools. Usually the objects protected by these permissions will receive their initial values during the X server startup process, and not be modified again.

Table 10. Permissions for the Server object classes

Permission(s)	Description
screensaver	Manage the screen saver
hostcontrol	Control host access
setfontpath	Set the font search path
getext	Query or list extensions
getattr	Get attributes of the server

5.2. Control Requirements

In this section control requirements are now defined for each X protocol request that manipulates the objects managed by the X server. The control requirements specify the permissions that must be granted for the request to successfully execute.

In the following tables, the control requirements for each request are specified, where each control requirement is described by the class, permissions, source SID (SSID), and target SID (TSID) used in a permission check. Since multiple requests may have the same requirements, more than one request may be listed in the leftmost column of a single table entry. In this case, all of the requirements in that table entry apply to all of the requests.

5.2.1. Control Requirements for Drawables

The requests that reference objects of the Drawable object class are mostly draw functions. The only permission that they need to checked is the permission to draw to that object. The `CopyArea`, `CopyPlane`, and `GetImage` requests can also copy the contents of a Drawable, so these methods will also check for this permission.

The `GetGeometry` request returns the geometry of the Drawable, which could have been either a Window or a Pixmap. The X server will only check the permissions relevant to the actual object type.

The `PolyText8`, `PolyText16`, `ImageText8`, and `ImageText16`, requests are listed in Table 15 with other font requests.

Table 11. X Server Drawable Object Control Requirements

Operation	Class	Permission	Source SID	Target SID
ClearArea FillPoly PolyArc PolyFillArc PolyFillRectangle PolyLine PolyPoint PolyRectangle PolySegment PutImage	Drawable	draw	client	window.drawable
CopyArea CopyPlane	Drawable	draw copy	client	dst.drawable src.drawable
GetImage	Drawable	copy	client	drawable
GetGeometry	Drawable	getattr	client	drawable

5.2.2. Control Requirements for Windows

As windows are the base input and output object for most X applications, the control requirements for the Window object class are extensive.

The entry in the control table for `CreateWindow` is abbreviated. The `CreateWindow` request takes an array of attributes which it uses to set the initial values for the window. This array is the same array that is passed to `ConfigureWindow`, and so the same permission checks will need to be performed.

Table 12 covers many of the situations in which event permissions will be checked, but not all of them. The event permissions will be checked whenever an event of the appropriate type is sent to the client. This may be as direct result of a client request, in which case the permissions are listed with the request, or it may be more indirect. If the event was spawned directly from a client request, as in the case of the `SendEvent` and the `SelectionClear` requests, the SSID will be the SID of the client. Otherwise, the SSID will be the SID of the X server. For input events, the SID will be the SID of the Input object, a second virtual singleton object. The label of the Input object will initially be the same as the server's SID, but may be changed by a trusted window manager. The target SID checked is the SID of the window which is nominally the target of the event.

Table 12. X Server Window object Control Requirements

Operation	Class	Permission	Source SID	Target SID
ChangeProperty	Window	chprop chproplist listprop	client client client	window window window
ChangeSaveSet	Window	ctrllife chparent	client client	window window
ChangeWindowAttributes	Window	setattr getattr	client	window
CirculateWindow	Window	chstack	client	window
ConfigureWindow	Window Window Window Window Window Window Cursor	setattr move chstack windowchangeevent windowchangeevent windowchangerequest assign	client client client client client client client	window window window window window.parent window.parent cursor
ConvertSelection	Window	clientcomevent	client	selection.owner
CreateWindow	Window Window Window Window Window Window Drawable	create setattr chstack move addchild windowchangeevent create	client client client client client client client	window window parent window parent parent window
DeleteProperty	Window	chprop chproplist	client client	window window
DestroyWindow DestroySubwindows	Window Window Window Drawable Window	enumerate unmap destroy destroy windowchangeevent	client client client client client	window twindow twindow twindow.drawable twindow

Operation	Class	Permission	Source SID	Target SID
GetMotionEvents	Input	mousemotion	client	window
GetProperty	Window	listprop	client	window
GetWindowAttributes	Window	getattr	client	window
KillClient	Client	kill	client	resource.owner
ListProperties	Window	listprop	client	window
MapWindow	Window Window Window Window Drawable	getattr windowchangeevent windowchangeevent drawevent copy	client client client client client	window window window.parent window coveredwindow
MapSubwindows	Window Window Window Window Window Drawable	enumerate getattr windowchangeevent windowchangeevent drawevent copy	client client client client client	window window.child window.child window window window.child coveredwindow
QueryTree	Window	enumerate	client	window
RotateProperties	Window	chproplist windowchangeevent	client	window
ReparentWindow	Window	addchild chparent move windowchangeevent windowchangeevent windowchangeevent	client client client client client client	parent window window window parent event
SendEvent	Window	clientcomevent inputevent drawevent windowchangeevent windowchangerequest serverchangeevent	client	window
SetInputFocus	Window Input	inputevent grab	client client	window screen
SetSelectionOwner	Window	chselection chselection clientcomevent	client client client	selection.window window selection.window
TranslateCoordinates	Window	getattr getattr	client client	source destination

Operation	Class	Permission	Source SID	Target SID
UnmapWindow UnmapSubwindows	Window	enumerate getattr windowchangeevent windowchangeevent	client client client client	window window window window.parent
WarpPointer	Input Window Window Window	grab inputevent inputevent getattr	client client client client	input srcwindow destwindow srcwindow

5.2.3. Control Requirements for Input

Sometimes an X client wishes to prevent other clients from receiving input from the keyboard or the mouse. For example, an application wishing to see if a click is a click or a drag will need to compare button press and release events. This is much easier if it can prevent other applications from receiving pointer events until drag is finished. The X server implements “grab” functionality to support these applications.

By calling `GrabKeyboard` or `GrabPointer`, the application immediately grabs the input from the keyboard or the pointer, redirecting all future input to the client until the grab is released. The `GrabKey` and `GrabButton` calls specify conditions when a grab will occur. Pointer grabs are used to generate system modal dialogs, a practice of declining popularity. Keyboard grabs are rarely used, but the ssh-ask graphical application for grabbing the pass-phrase for ssh uses it to prevent keyboard snooping.

A malicious application can currently change the keyboard mappings in order to disrupt the user. Changes to the input settings are closely related to user preferences, so prudent policies will limited these changes to direct action from the user.

As the keyboard keymap is shared by all applications, the context of the single server object will be used on permission checks. Like other server parameters, these settings may be changed during the server initialization, but generally won’t be during normal use. This may not be the case with international users, however. Both KDE and GNOME come with mini applications to change the keyboard mappings, primarily targeted at users of international keyboards.

Applications need to query the state of they keyboard to translate the keycode in a ButtonDownEvent with the character that keycode represents. Very few policies will want to restrict the ability of a client to query the keyboard state.

Table 13. Input Control Requirements

Operation	Class	Permission	Source SID	Target SID
GrabButton GrabKey	Input	passivegrab	client	server
GrabKeyboard GrabPointer	Input	grab	client	screen

Operation	Class	Permission	Source SID	Target SID
AllowEvents UngrabButton UngrabKey UngrabKeyboard UngrabPointer	Input	ungrab	client	server
GetKeyboardControl GetPointerControl GetPointerMapping GetModifierMapping QueryKeymap QueryPointer	Input	getattr	client	Server
ChangeKeyboardControl ChangePointerControl SetModifierMapping SetPointerMapping	Input	setattr	client	Server

5.2.4. Control Requirements for Colors and Colormaps

The control requirements for X protocol operations that manage Color and Colormaps objects are described in this section.

Table 14. Color and Colormap Control Requirements

Operation	Class	Permission	Source SID	Target SID
AllocColor	Color Colormap	create read store	client client color	color colormap colormap
AllocColorCells	Color Colormap	create read store	client client color	color colormap colormap
AllocColorPlanes	Color Colormap	create read store	client client color	color colormap colormap
AllocNamedColor	Color Colormap	create read store	client client color	color colormap colormap
CopyColormapAndFree	Colormap	create read free	client client client	new colormap source colormap source colormap
CreateColormap	Colormap Drawable	create draw	client client	colormap window.draw
FreeColormap	Colormap	free	client	colormap

Operation	Class	Permission	Source SID	Target SID
FreeColors	Color Colormap	free store	client client	color colormap
InstallColormap	Colormap	install	client	server
ListInstalledColormaps	Colormap	list	client	server
LookupColor	Color	lookup	client	colormap
QueryColors	Colormap	read	client	colormap
StoreColors	Colormap	store	client	colormap
StoreNamedColor	Colormap	store	client	colormap
UninstallColormap	Colormap	uninstall	client	colormap

5.2.5. Control Requirements for Fonts and Text

The control requirements for X protocol operations that manage Font objects and text manipulation are described in this section.

The Font objects are centralized in the server for performance reasons. An application lists the fonts, loads the fonts it wants to use, and then uses the font to draw into a Drawable. The SID for a Font object will be derived from the label of the font source file, allowing the policy to make font-specific decisions.

Table 15. Font and Text Control Requirements

Operation	Class	Permission	Source SID	Target SID
CloseFont	Font	free	client	font
GetFontPath	Font	getattr	client	server
ImageText8 ImageText16	Font Drawable	use draw	client client	font drawable
ListFonts ListFontsWithInfo	Font	getattr	client	server
OpenFont	Font	load use	client client	server font
PolyText8 PolyText16	Font Font GContext Drawable	load use chattr draw	client client client client	server font gc drawable
QueryFont	Font	getattr	client	font
QueryTextExtents	Font	getattr	client	font

5.2.6. Control Requirements for Pixmaps

A Pixmap object is normally used by applications to cache the results of drawing operations. The client will initially draw some complex entity to a pixmap, then copy that pixmap to the target window. Reading a client's pixmap is therefore equivalent to reading the data from the client's window, and writing to the pixmap may be equivalent to writing to the client's window.

While a Window object is more than just a Drawable object, the same can not be said for a Pixmap; thus Pixmap object in the server is represented in the policy only by Drawable objects. Creating a Pixmap creates a Drawable. The Drawable's context is derived from the context of the client that creates it.

Table 16. Pixmap Control Requirements

Operation	Class	Permission	Source SID	Target SID
CreatePixmap	Drawable	create	client	pixmap
FreePixmap	Drawable	free	client	pixmap
CreateCursor	Drawable	draw	client	pixmap

5.2.7. Control Requirements for the Cursor object

Cursor objects are associated with windows. When the pointer is over a window, the Cursor object associated with that window will be shown as the pointer image. Most applications create their cursors from a standard cursor font loaded by the server. To limit an application to the standard cursors, the policy should deny the `create` permission and limit the use of fonts.

Table 17. Control requirements for the cursor

Operation	Class	Permission	Source SID	Target SID
CreateCursor	Cursor	create	client	cursor
CreateGlyphCursor	Cursor Font	createglyph use	client	cursor font
RecolorCursor	Cursor	setattr	client	cursor

5.2.8. Control Requirements for the Server object

The Server object does not represent any object actually managed by the X server. Instead, it is a virtual object used to provide a label for properties of which are global to the X server, and not specific to any one client.

The screen saver requests also need to be enforced on the requests in the `MIT-SCREEN-SAVER` extension. Those control requirements are not listed here, as this paper has not addressed the control requirements of any extensions.

The host-based access controls secured by the `hostcontrol` permission have generally been deprecated. The replacement, MIT magic cookies, are not managed through the X server, and therefore can not be controlled by X server permissions.

Table 18. Control requirements for the server

Operation	Class	Permission	Source SID	Target SID

Operation	Class	Permission	Source SID	Target SID
ForceScreenSaver GetScreenSaver SetScreenSaver	Server	screensaver	client	server
ChangeHosts SetAccessControl	Server	hostcontrol	client	server
SetFontPath	Server	setfontpath	client	server
ListExtensions QueryExtension	Server	getext	client	server
QueryBestSize	Server	getattr	client	server

5.2.9. Extensions

To permit applications to create objects with a specified label rather than the default label, an extended form of each of the object creation requests must be added that accepts an additional context parameter. To permit clients to obtain the context of an object, an extended form of each of the object status requests must be added that returns an additional context parameter. To permit clients to change the context of an object, new requests must be added for these security-aware applications and are shown in Table 19.

For the new requests that are extended forms of existing requests, the existing set of control requirements apply.

Table 19. Control requirements for relabeling

Operation	Class	Permission	Source SID	Target SID
RelabelWindow	Window	relabelfrom	client	window
RelabelInput	Input	relabelinput	client	input

5.3. Events

Events are sent asynchronously to clients, sometimes as the result of a request but usually as the result of user actions. Events include notification that the mouse moved, notification that a portion of the window needs to be redrawn, or messages passed between clients. There are 36 different events in the base X protocol, which are grouped into 6 permissions in the Window object class. These groups are shown in Table 20.

To avoid duplication, only the permission to send an event is shown in the table. The source SID will be the source of the event. For example, for real input events the source context will be input object. The target SID will be the owner of the window being targeted by the event.

While these permissions are listed under `SendEvent` and other requests, in fact there are many ways events can be sent to windows. The control requirements tables list event permissions when a request could generate an event, but events generated by the server are not listed. The server will need to be modified to always check for permission to send an event. The permission to register for events is not covered by this framework. Instead the policy is expected to enforce the event permissions.

The type of event is important to any policy. A well-behaved client that claims the clipboard will need to be able to send a SelectionClear event, to let the client that currently owns the clipboard know about the change. However, no well-behaved client will send a KeyPress event. Policies that seek to enforce good behavior should limit the types of events that clients can generate.

Table 20. Event Permissions

Events	Event Permission	Description
SelectionClear SelectionNotify SelectionRequest ClientMessage PropertyNotify	clientcomevent	Events for inter-client communication
ButtonPress ButtonRelease KeyPress KeyRelease KeymapNotify MotionNotify EnterNotify LeaveNotify FocusIn FocusOut	inputevent	User Input Events
Expose GraphicsExpose NoExpose VisibilityNotify	drawevent	Events telling the client to redraw
CirculateNotify ConfigureNotify CreateNotify DestroyNotify MapNotify UnmapNotify GravityNotify ReparentNotify	windowchangeevent	Events informing the client of changes to a window
CirculateRequest ConfigureRequest CreateRequest DestroyRequest MapRequest ResizeRequest SubStructureNotify	windowchangerequest	Events to the Window Manager requesting changes to a window
ColormapNotify MappingNotify	serverchangeevent	Events telling the client about global server changes

5.3.1. Client Communication Events

Client communication events exist to enable client communication, and will be sent by clients to other clients. While any event can be sent via `SendEvent` the expectation is that all other event types are only generated by the server.

The SelectionClear, SelectionNotify, and SelectionRequest events are used to notify clients of changes to selection buffers used for cut-and-paste functionality. When a client wishes to export a selection buffer, it will use the `SetSelectionOwner` request, which will generate the SelectionClear event to the previous owner of the selection. When a client wants to get the selection in a different format, it calls the `ConvertSelection` request, which causes the server to send a SelectionRequest event to the owner of the selection. The owner of the selection is expected to convert the selection to the requested type, and send a SelectionNotify event to the requesting client by the `SendEvent` request.

The ClientMessage event is an arbitrary event. Two clients who have agreed on the interpretation of data can exchange ClientMessage events. The X server will never generate this type of event.

The PropertyNotify event is sent to a client when properties on a window change. Since the server never changes a windows properties, this event results from client communication.

A policy should only allow clients to send these events to each other, or cause these events to be sent to each other, if it would otherwise allow direct communication between the clients.

5.3.2. Input Events

Input events should only be sent by the server. Input events are generated by the user's interactions with the input hardware. They tell the client about input, and alert the client when input will be directed to the client.

The ButtonPress and ButtonRelease events inform the client when any buttons on the pointing device have been pressed or released. The client can also find out the state of the modifier keys on the keyboard as part of these events. The MotionNotify event gives the client information about the motions of the mouse pointer. A MotionNotify event is guaranteed to be generated only if the motion begins or ends inside the window.

The KeyPress and KeyRelease events inform the client when any keys on the keyboard have been pressed or released. There is no way for a client to select event notification on a subset of the keyboard. If the client receives key events it will receive them when any key on the keyboard is pressed, even the modifier keys.

The KeymapNotify event is reported to clients when a window acquires keyboard input focus. The event allows the client to determine which keys were pressed when the input focus was transfered to the window.

The EnterNotify and LeaveNotify events inform the client when the mouse pointer has entered or left a window. The FocusIn and FocusOut events inform the client when the window has keyboard input focus.

5.3.3. Draw Events

These events alert the client that some portion of its window needs to be redrawn. The Expose event gives the client a list of rectangular areas that need to be redrawn in the given window. In normal operation this event, like most, will only be generated by the server.

The VisibilityNotify event can be used to alert clients that a section of their window is no longer visible. A client could use this information to avoid updating that portion of their window when it changes, for example. The event will also be issued when the window regains visibility.

The NoExpose and GraphicsExpose events are generated by `CopyArea` and `CopyPlane` requests. If the copy failed because the source region is obscured, unmapped, or otherwise unavailable, a GraphicsExpose event will be generated. Otherwise, a NoExpose event will be generated.

5.3.4. Window Change Events

These events alert a client when the window manager performs various window operations. They originate only from the X server. There will rarely be any need to prevent a client from receiving any of these events, as they only inform the client of operations that have already occurred. The majority of these events can be sent to both the window that was modified, and the parent of that window.

The CirculateNotify event informs a client when the stacking order has changed. This is normally related to focus changes. The ConfigureNotify event is sent to the client when a window has had its configuration changed by `ConfigureWindow`.

The CreateNotify and DestroyNotify events are sent to a client when a sub window is created or destroyed. The MapNotify and UnmapNotify events are sent to the window being mapped and unmapped. The GravityNotify event is sent to a window when it has moved because of a change in size of its parent. The ReparentNotify event is sent to a window that has been re-parented.

5.3.5. Window Change Request Events

These events are used by the Window Manager to control certain window manager functionality. If a client requests an operation that the Window Manager controls, the X server will send an appropriate event to the window manager instead of performing the operation. The Window manager can then determine if the operation should be performed, and if so, tell the X server to perform the operation.

Only the window manager should be allowed to to select the `StructureRedirectMask` events on the root window, and clients should not be allowed to set the `override_redirect` attribute on their top-level windows. Some older applications may set the attribute for transient windows, but more recent applications will generally use window hints instead. Applications which set the attribute will generally work with the attribute unset.

The different events covered by the `windowchangerequest` permission contain the same information, and have the same meaning as the equivalent event listed under the `windowchangeevent` permission. Those permissions are discussed in Section 5.3.4.

5.3.6. Server State Change Events

These events are sent to clients when global server state changes. These events should only be sent from the server, although they will frequently result directly from client requests. These events should generally not be quelled by the policy.

The ColormapNotify events is reported to clients when the colormap attribute of a window is changed, or the currently installed colormap changes. Clients that wish to correct their colors based on the currently installed colormap need to receive this event.

The MappingNotify event is reported to all clients when the mappings for the input devices change. Clients need to receive this event to keep their model of the keyboard and pointer mappings consistent with the X server's model.

5.3.7. Event Control Requirements

The control requirements for events generated as the result of client requests are shown in the client request tables. Events can also be dispatched directly from the server, particularly in the case of input events. Any client can forge any event through the `SendEvent`. Events sent through `SendEvent` will always have the source SID of the sending client, and the target SID of the target window.

The control requirements for events covered by the `clientcomevent` are included in Table 12. The source SID and target SID used are listed there. The target will always be the window receiving the event, and the source will be the client sending the event, or the client that invoked the request that directly caused the event to be sent.

By itself, applying a policy to the `inputevent` permission will not enforce Trusted Path functionality, as there is not sufficient information to determine the window which has focus. As previously mentioned, a trusted Window Manager could apply a label to the input. The source SID for events covered by the `inputevent` permission will be that of the input object. Applications will also need to carefully grab input when required, and policies will need to prevent malicious applications from grabbing the input.

For events covered by the `serverchangeevent`, the source SID will be that of the server object, even though these events happen in response to actions of the client. Clients need to receive these events to properly understand the state of the server, yet these events are indirectly generated by clients. These events allow a high-bandwidth covert channel, and so the permissions to install colormaps or change keyboard mappings should be tightly regulated.

The events covered by the `drawevent` permission are considered to be asynchronous events. The source SID will be that of the server object. Like the `serverchangeevent` events, Expose and VisibilityNotify events can result from client requests, and so could be used for covert channels. When the expose event is the result of client actions the server should modify the information in the event to reduce the information passed. The server should enlarge the exposed rectangle to include the entire window.

The events covered by the `windowchangerequest` permission should only be sent to the window manager. The events are generated as the result of client requests, so the source SID will be that of the client, and the target SID will be that of the window manager. The events covered by the `windowchangeevent` permission will use the source SID of the client performing the operation, which should be that of the window manager.

6. Implementation

Many of the checks that will be required have been added to the XFree X server by the X11 Security Extension. While the Security Extension does not fully label both subjects and objects, it does do a certain amount of labeling, and defines an API that is widely used in the X server for the checks.

Unfortunately, the Security Extension does not appear to be fully implemented at the time of this report. Only four types of access (actions) are recognized, one of which is Unknown_Access, and

Unknown_Access is always permitted. The degree of effort required to expand the extension to support a richer policy is unknown.

6.1. Enforcing Permissions on Requests

All communication between an X client and the X server happens over the protocol. Performing mediation at the X protocol interface is sufficient to ensure that clients can not communicate via the server without authorization. Extensions like the MIT-SHM (the MIT Shared Memory Extension) merely allow clients to share image data with the server, and do not present additional channels of communication.

One place to put the hook for all requests is ProcessWorkQueue (dix.c) in the XFree86 source tree, the central dispatch loop for the X server. However, it is likely that each request will need to be modified to check permissions, as the central dispatch loop does not know the data contained in the request.

6.2. Enforcing Permissions on Events

For events, there are several control points. Some events are sent via `WriteToClient`, a low level output method. Some events are sent via `DeliverEvents`, some by `MaybeDeliverEventsToClient`, and most by `WriteEventsToClient`.

Since there is no single interface for intercepting the events being sent to a client, the X server will need to be modified to be more consistent about sending events to a client. The X server currently treats events like any other bit of opaque data that needs to be sent to the client: it queues the event onto an output buffer. While one could certainly place a hook in the low level `WriteToClient` method, the hook will need to implement a significant amount of logic to determine the type of the opaque data being queued.

6.3. Extensions

Another feature that needs access control is the use of extensions. The X protocol is flexible, and allows arbitrary extensions to be registered. It is not practical for an ongoing security project to know the security properties of a new extension, so part of the policy will need to limit extensions. While extensions could be aware of the security policy, they do constitute code running inside a trusted enforcement module. Limiting extensions should be possible from the kernel-level policy for the X server, as the extensions are implemented as shared libraries that the X server links in at run time.

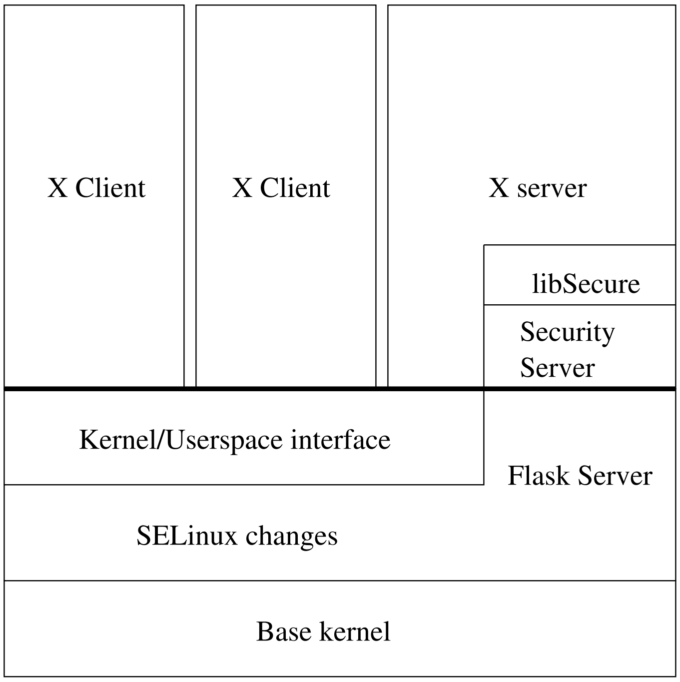
6.4. Interactions with Flask

The SELinux framework currently trusts only the kernel to make security decisions, and trusts only the kernel to enforce security decisions. However, it is not practical to refactor the X server to allow enforcement to take place in the kernel. The X server will need to enforce the policy.

Policies for X objects will likely be incorporated with the general system policy. If two processes should not communicate by more traditional system means like Unix domain sockets, then they should also be prevented from communicating through the X server. If one application's output should not be allowed as input to a different application, then that second application should not be allowed to copy a window belonging to the first application.

The X server should use the Access Vector Cache (AVC) component from the Flask architecture. The AVC component is not currently included in libSecure, and will need to be. The AVC will need a security server to make policy decisions. The security server could be included in libSecure, or the AVC could use the security server in the kernel. Both approaches are valid, but we recommend initially using the kernel's security server. Using the security server already in SELinux will consolidate the security server code and simplify the process of changing the current policy. If the performance of this approach is unacceptable, the AVC architecture would make moving the security server into the X server straight-forward.

Figure 2. Interface Diagram



The diagram in Figure 2 represents the classical layer diagram of the proposed design. The X server will use the AVC implementation in libSecure for authorization, which in turn will use a security server. The X server will rely on the SELinux controls for protection against other processes. The clients will communicate with the server through TCP/IP connections or Unix Domain Sockets via the normal X protocol, where the server will enforce the systems policy upon the client requests.

6.5. Handling Errors

The core X protocol defines 17 errors; some are generic (e.g. `Access`). Other errors are specific to a request; `BadPixmap` for example. Most of these errors are returned to a client when the client provides bad arguments to a request.

When a client attempts to perform an operation that is not permitted by the security policy, the X server must decide how best to respond. Depending on the severity of the error, there are at least three options. The X server can ignore the request, send a protocol error to the client, or terminate the client. The SELinux security server typically returns errors that only have meaning within the Linux kernel; these error values must be translated into something that is more appropriate for X. Some probable translations are listed in Table 21. The actual error, if any, returned when a request is denied will be dependent on the request, and the exact reason why a request was denied. For example, if a client invokes the `QueryTree` request, and some child window is not visible to the client, the child window should just be eliminated from the output and no error sent. If the client attempts to set a property on a window that should not be visible (by guessing its Resource ID), then a `BadWindow` error should be sent.

Table 21. Translation table for SELinux error codes

SELinux error code	X Error
ENOENT	BadColormap BadCursor BadDrawable BadFont BadPixmap BadValue BadWindow
EAGAIN	no direct translation
EACCESS	BadAccess
EPERM	BadAccess
EEXIST	BadIDChoice
EINVAL	BadValue

If the error reflects an intermittent failure within the security server (ENOMEM), it may be preferable to report the error to the application and allow the user to potentially retry the operation. In this case, ENOMEM would translate to a `BadAlloc` error condition. The X protocol already defines a `BadAccess` error that may be returned for some operations. However, a `BadAccess` error is not an expected result of many operations and there may be unexpected consequences for the client application.

The `EAGAIN` error indicates that the system has a temporary issue, and that the client should try the request again. In most cases, this return code indicates a transient issue with the security server, and the modified X server will rerun the query instead of passing the error back to the client.

It will often be the case that most operations can simply be stopped. This may cause problems with some clients, since few X protocol methods are expected to fail. Even considering the impact upon client applications, it is preferable to err on the side of stronger security. Even though clients may freeze as a result of access denials, the X server may not need to terminate the application. [WigginsProtocol96]

7. Security-Aware Applications

In this report, we have assumed that the vast majority of X applications will be unmodified, traditional, security-oblivious applications. None of the preexisting base of X applications are aware of security enhancements. However, there will be some applications which will need or want awareness of the security policy they are running under. An obvious example would be a large office application, which would want to execute document macros in the context of the document. While a graphical login program will need to be security aware, we do not expect it will need to be aware of the X security policies.

7.1. The Window Manager

As we have mentioned throughout the report, the design of the security-aware X server assumes the existence of a trusted window manager. Certain security decisions need input that the X server can not provide, but the window manager can. By relying on the window manager to make proper visual labeling decisions, we can express these policies in the general policy language.

The server will have a single labeled object to represent the input stream. As mentioned in Section 5.2.3, the window manager is responsible for labeling this object appropriately for the window which has input focus.

Most current window managers provide visual labels to let the user know which window has keyboard focus. Client applications provide desired strings for a title bar. The window manager could also visually label windows according to their security context, if this functionality was desired.

Linux systems may use a variety of window managers, all with slightly different functionality. Fortunately, KDE and GNOME have become the defacto standards for desktop environments, and either desktop works when using the other desktop's window manager. By modifying the either of the two window managers to supports the security extensions, we will get suitable coverage for a large portion of the Linux user base.

7.2. Large Integrated Applications

Large integrated applications, like StarOffice/OpenOffice present unique issues for security. A single application may have several documents open at the same time, and so be managing multiple security domains. These large office applications also include sufficient macro functionality to be considered a run-time environment instead of merely being a document editor and viewer.

While no one has yet started making these large applications security aware, we expect that future work will need to consider their needs. These applications will also need to be aware of the security policies of the X server, if just to help properly direct input.

7.3. Other Applications

The free desktop environments will also need to be security aware, but may not be aware of any X related security policies. While you want your file browser to be able to show you the extended security information of your files, it should not need special handling by the X server when it does so.

The file browsing functionality of KDE's file manager and the GNOME equivalent may be crossing security domains, but at least in the KDE case, the file viewer application is a separate program, and can run in a separate domain. To handle these applications, the policy needs to handle applications with sub-windows belonging to other applications, but this should be possible without needing help from the parent application.

8. Conclusions

Having recognized the need for additional security for commonly used and freely available operating systems, the NSA undertook development of Security Enhanced Linux. By integrating strong, flexible mandatory access control into Linux, overall system security has been improved. However, there is still more work that can be done to secure the user operating environment while still providing a usable workstation. By extending the SELinux security server to user-space applications such as the X Server, it becomes easier to apply a consistent security policy to the system as a whole.

By modifying the SELinux policy to place the X11 Server into a trusted domain, and by further permitting the Server to enforce policy upon X clients, it is possible to allow uses to run a graphical environment and still provide system integrity and to restrict information flow. Through a careful examination of the X protocol, this report describes which objects must be secured and which types of operations may be performed upon them. In this way, all communication between an X server and X clients can be intercepted and a security policy may be enforced.

This document defines the set of object classes within X11 that must be labeled and controlled in order to provide reasonable security over the X protocol. While the lists of permissions for these objects are quite extensive, after further experimentation and testing, it may be reasonable to combine some similar permissions and yield a smaller effective permission set.

Most X Server implementations allow for modular extensions to be loaded so that the Server functionality can be enhanced. This document does not address the issue of X Server extensions, but due to their similarity to Linux kernel modules, they can likely be handled in a similar manner. Like kernel extensions, it is difficult to know the exact contents of an extension, and what affect it will have on the system. For this reason, the decision to support modules and extensions is often binary; either all extensions are permitted or not.

References

[ClipExt] Lubos Lunak, *Proposal for Clipboard Extensions*, https://listman.redhat.com/pipermail/xdg-list/2002-November/000881.html , November 2002.

[FlaskArch] Ray Spencer, Peter Loscocco, Stephen Smalley, Mike Hibler, David Anderson, and Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies", *Proceedings of The Eighth USENIX Security Symposium*, The USENIX Association, August 1999.

[Kirk97] Kirk Bittler, "A Policy-Independent Secure X Server", A Thesis Submitted To The Master of Science in Computer Science Program at Portland State University, 1997.

[LoscoccoFreenix2001] Peter Loscocco and Stephen Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System", *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, The USENIX Association, June 2001.

[LoscoccoNSATR2001] Peter Loscocco and Stephen Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System", *NSA Technical Report*, February 2001.

[LoscoccoNISS1998] Peter Loscocco, Stephen Smalley, Patrick Muckelbauer, Ruth Taylor, S. Turner, and John Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern

Computing Environments", *Proceedings of the 21st National Information Systems Security Conference*, October 1998.

[OReilly90] Edited by Adrian Nye, "X Protocol Reference Manual, Second Edition", O'Reilly & Associates, Inc., 1990.

[SELinux] The National Security Agency, *Security-Enhanced Linux*.

[SpencerUsenixSec1999] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies", *Proceedings of the Eighth USENIX Security Symposium*, The USENIX Association, August 1999.

[WigginsProtocol96] David Wiggins, "Analysis of the X Protocol for Security Concerns: Draft Version 2", The X Consortium, Inc., May 10, 1996.

[WigginsExtension96] David Wiggins, "Security Extension Specification Version 7.1 X11 Release 6.fi", The X Consortium, Inc., November 15, 1996.

[Xlib] Adrian Nye, "The Definitive Guides to the X Window System", Volume One, *Xlib Programming Manual for Version 11*, O'Reilly & Associates, Inc., 1989.

[Xlib2] Jim Gettys and Robert Scheifler, "Xlib"*: C Language X Interface*, The X Consortium, Inc., 1996.

[XlibRef] Jim Gettys, Ron Newman, and Robert Scheifler, Edited by Adrian Nye, "The Definitive Guides to the X Window System", Volume Two, *Xlib Reference Manual for Version 11*, O'Reilly & Associates, Inc., 1989.

[XOrgIntro] *Introduction: About the X Window System*, Available at http://www.x.org/X11.html.

[XProtocol] Robert Scheifler, "X Window System Protocol: X Version 11, Release 6.1", The X Consortium, Inc., 1994.

[XSelections] Keith Packard, "The X Selection Mechanism"*: How to Cut and Paste in 1000 Lines or More*, The X Consortium, Inc., Laboratory for Computer Science, Massachusetts Institute of Technology, 1990.