X Access Control Extension Specification

Eamon F. Walsh

2009

Revision History

Revision	Date	Author
Revision 1.0	19 Oct 2006	efw
	Initial Version	
Revision 2.0	10 Mar 2008	efw
	Version 2.0	
Revision 2.1	19 Jun 2009	efw
	Version 2.1 (XI2)	
Revision 2.2	29 Jun 2009	efw
	Version 2.2 (Property post-data hook)	

Abstract

The X Access Control Extension (XACE) is a set of generic "hooks" that can be used by other X extensions to perform access checks. The goal of XACE is to prevent clutter in the core dix/os code by providing a common mechanism for doing these sorts of checks. The concept is identical to the Linux Security Module (LSM) in the Linux Kernel.

XACE version 1.0 was a generalization of the SECURITY extension, which provides a simple on/off trust model where "untrusted" clients are restricted in certain areas. Its hooks were for the most part straight replacements of the old SECURITY logic with generic hook calls. XACE version 2.0 has substantially modified many of the hooks, adding additional parameters and many new access types. Coverage has also been extended to many additional extensions, such as Render and Composite. Finally, there is new support for polyinstantiation, or duplicate, window properties and selections.

This paper describes the implementation of XACE version 2.0, changes to the core server DIX and OS layers that have been made or are being considered, and each of the security hooks that XACE offers at the current time and their function. It is expected that changes to XACE be documented here. Please notify the authors of this document of any changes to XACE so that they may be properly documented.

Table of Contents

Introduction

Prerequisites

This document is targeted to programmers who are writing security extensions for X. It is assumed that the reader is familiar with the C programming language. It is assumed that the reader understands the general workings of the X protocol and X server.

Purpose

XACE makes it easier to implement new security models for X by providing a set of pluggable hooks that extension writers can use. The idea is to provide an abstraction layer between security extensions and the core DIX/OS code of the X server. This prevents security extensions writers from having to understand the inner workings of the X server and it prevents X server maintainers from having to deal with multiple security subsystems, each with its own intrusive code.

For example, consider the X.Org X server's resource subsystem, which is used to track different types of server objects using ID numbers. The act of looking up an object by its ID number is a security-relevant operation which security extension writers would likely wish to control. For one or two security extensions it may be acceptable to simply insert the extension's code directly into the resource manager code, bracketed by `ifdef`'s. However for more extensions this approach leads to a tangle of code, particularly when results need to be logically combined, as in `if` statement conditions. Additionally, different extension writers might place their resource checking code in different places in the server, leading to difficulty in tracking down where exactly a particular lookup operation is being blocked. Finally, this approach may lead to unexpected interactions between the code of different extensions, since there is no collaboration between extension writers.

The solution employed by the X Access Control Extension is to place hooks (calls into XACE) at security-relevant places, such as the resource subsystem mentioned above. Other extensions, typically in their initialization routines, can register callback functions on these hooks. When the hook is called from the server code, each callback function registered on it is called in turn. The callback function is provided with necessary arguments needed to make a security decision, including a return value argument which can be set to indicate the result. XACE itself does not make security decisions, or even know or care how such decisions are made. XACE merely enforces the result of the decision, such as by returning a BadAccess error to the requesting client.

This separation between the decision-making logic and the enforcement logic is advantageous because it allows a great variety of security models to be developed without resorting to intrusive modifications to the core systems being secured. The challenge is to ensure that the hook framework itself provides hooks everywhere they need to be provided. Once created, however, a hook can be used by everyone, leading to less duplication of effort.

Prior Work

Security Extension

XACE was initially based on the SECURITY extension. This extension introduced the concept of "trusted" and "untrusted" client connections, with the trust level established by the authorization token used in the initial client connection. Untrusted clients are restricted in several areas, notably in the use of background "None" windows, access to server resources owned by trusted clients, and certain keyboard input operations. Server extensions are also declared "trusted" or "untrusted," with only untrusted extensions being visible to untrusted client connections.

Solaris Trusted Extensions

Trusted Extensions for Solaris has an X extension (Xtsol) which adds security functionality. Some of the XACE hooks in the current set were derived from security checks made by the Xtsol code. In other places, where the Xtsol and SECURITY extensions both have checks, a single XACE hook replaces both.

Linux Security Modules

XACE is influenced by the Linux Security Modules project, which provides a similar framework of security hooks for the Linux kernel.

Version 2.0 Changes

Different Return-Value Semantics

The status value returned by security modules has been changed. Formerly, security modules were expected to set the "rval" field of the input structure to "False" if access was to be denied. In version 2.0, this field has been removed in all hooks. Security modules must now set the "status" field to an X error code to describe the error. Typically, `BadAccess` will be returned, but this change allows security modules to return `BadAlloc` to report memory allocation failure and `BadMatch` to report a polyinstantiated object lookup failure (the section called “Polyinstantiation”).

DevPrivates Mechanism

The devPrivates mechanism in the X server was substantially revised to better support security extensions. The interface for using devPrivates has been unified across the different structures that support private data. Private space allocation is now independent of whether objects have already been created, and the private indexes are now global rather than being structure specific. Callbacks are available to initialize newly allocated space and to clean up before it is freed. Finally, there is a mechanism for looking up the offset of the private pointer field in a structure, given the structure's resource type.

New Access Modes

In the previous version, there were four possible modes for the "access_mode" field: read, write, create, and destroy. In version 2.0, many new modes have been introduced to better describe X operations, particularly on window objects. The access_mode field has also been added to additional hooks as described in the individual hook changes.

Polyinstantiation

XACE now supports polyinstantiation of selections and window properties. the section called “Property Access” and the section called “Selection Access” describe the details, but the basic idea is that the property and selection access hooks may be used to not only change the return value of a lookup operation but also to modify the lookup result. This allows more than one property or selection with the same atom name to be maintained.

Removed Hooks

The "drawable," "map," "window init", and "background" hooks have been removed. They have been folded into the resource access hook using new access modes. The "hostlist" hook has been removed and replaced by a new server access hook (see the section called “Server Access”). The "site policy" and "declare extension security" hooks have been removed as the SECURITY extension has been revised to no longer require them.

New Hooks

New "send" and "receive" hooks have been added to allow basic control over event delivery. "Client" and "server" access hooks have been added to control access by clients to other clients (for example, through the `KillClient` call) and to the server (for example when changing the host access list or changing the font path). "Screen" and "screen saver" hooks have been added to control access to screens and screen saver requests. A "selection" hook has been added to control access to selections.

Changes to Existing Hooks

- The resource access hook structure now has additional fields to describe a "parent" object. They are set only when a resource with a defined parent (such as a Window object) is being created, in which case the access mode will include `DixCreateAccess`.

- The device access hook structure has had the "fromRequest" field removed and an access mode field added.

- The property access hook structure has had the "propertyName" field removed and a "ppProp" field added, which contains a pointer to a pointer to the property structure itself. The extra level of indirection supports polyinstantiation (see the section called “Polyinstantiation”). Note that the property structure contains the property name.

- The extension dispatch/access hook structure now has an access mode field.

Future Work

Security Hooks

It is anticipated that the set of security hooks provided by XACE will change with time. Some hooks may become deprecated. More hooks will likely be added as well, as more portions of the X server are subjected to security analysis. Existing hooks may be added in more places in the code, particularly protocol extensions. Currently, the only method XACE provides for restricting access to some protocol extensions is to deny access to them entirely.

It should be noted that XACE includes hooks in the protocol dispatch table, which allow a security extension to examine any incoming protocol request (core or extension) and terminate the request before it is handled by the server. This functionality can be used as a stopgap measure for security checks that are not supported by the other XACE hooks. The end goal, however, is to have hooks integrated into the server proper.

Core X Server

The set of extensions supported by X.org needs to be re-examined. Many of them are essentially unused and removing them would be easier than attempting to secure them. The GLX extension and the direct rendering kernel interfaces need to be secured.

The server's routines for event delivery need to be reworked to allow greater control by XACE modules. In particular, security extensions need to be able to associate private data with each event at the time of its generation based on the context and then have that data available at a decision point just before the event is delivered to the client. This would allow event delivery to be better controlled on a per-client basis, and would potentially allow additional security extension functionality such as piggyback events.

Usage

Storing Security State

The first thing you, the security extension writer, should decide on is the state information that your extension will be storing and how it will be stored. XACE itself does not provide any mechanism for storing state.

One method of storing state is global variables in the extension code. Tables can be kept corresponding to internal server structures, updated to stay synchronized with the structures themselves. One problem with this method is that the X server does not have consistent methods for lifecycle management of its objects, meaning that it might be difficult to keep state up to date with objects.

Another method of storing state is to attach your extension's security data directly to the server structures. This method is possible via the `devPrivates` mechanism provide by the DIX layer. Structures supporting this mechanism can be identified by the presence of a "devPrivates" field. Full documentation of the devPrivates mechanism is described in the core X server documentation.

Using Hooks

Overview

XACE has two header files that security extension code may need to include. Include `Xext/xacestr.h` if you need the structure definitions for the XACE hooks in your source file. Otherwise, include `Xext/xace.h`, which contains everything else including constants and function declarations.

XACE hooks use the standard X server callback mechanism. Your security extension's callback functions should all use the following prototype:

```
void MyCallback(CallbackListPtr *pcbl  pointer userdata
pointer calldata);
```

When the callback is called, *pcbl* points to the callback list itself. The X callback mechanism allows the list to be manipulated in various ways, but XACE callbacks should not do this. Remember that other security extensions may be registered on the same hook. *userdata* is set to the data argument that was passed to `XaceRegisterCallback` at registration time; this can be used by your extension to pass data into the callback. *calldata* points to a value or structure which is specific to each XACE hook. These are discussed in the documentation for each specific hook, below. Your extension must cast this argument to the appropriate pointer type.

To register a callback on a given hook, use `XaceRegisterCallback`:

```
Bool XaceRegisterCallback(int hook  CallbackProcPtr call-
back  pointer userdata);
```

Where *hook* is the XACE hook you wish to register on, *callback* is the callback function you wish to register, and *userdata* will be passed through to the callback as its second argument, as described above. See Table 1, “XACE security hooks.” for the list of XACE hook codes. `XaceRegisterCallback` is typically called from the extension initialization code; see the SECURITY source for examples. The return value is `TRUE` for success, `FALSE` otherwise.

To unregister a callback, use `XaceDeleteCallback`:

```
Bool XaceDeleteCallback(int hook  CallbackProcPtr callback
pointer userdata);
```

where the three arguments are identical to those used in the call to `XaceRegisterCallback`. The return value is `TRUE` for success, `FALSE` otherwise.

Hooks

The currently defined set of XACE hooks is shown in Table 1, “XACE security hooks.”. As discussed in the section called “Security Hooks”, the set of hooks is likely to change in the future as XACE is adopted and further security analysis of the X server is performed.

Table 1. XACE security hooks.

Hook Identifier	Callback Argument Type	Refer to
`XACE_CORE_DISPATCH`	XaceCoreDispatchRec	the section called “Core Dispatch”
`XACE_EXT_DISPATCH`	XaceExtAccessRec	the section called “Extension Dispatch”
`XACE_RESOURCE_ACCESS`	XaceResourceAccessRec	the section called “Resource Access”
`XACE_DEVICE_ACCESS`	XaceDeviceAccessRec	the section called “Device Access”
`XACE_PROPERTY_ACCESS`	XacePropertyAccessRec	the section called “Property Access”
`XACE_SEND_ACCESS`	XaceSendAccessRec	the section called “Send Access”
`XACE_RECEIVE_ACCESS`	XaceReceiveAccessRec	the section called “Receive Access”
`XACE_CLIENT_ACCESS`	XaceClientAccessRec	the section called “Client Access”
`XACE_EXT_ACCESS`	XaceExtAccessRec	the section called “Extension Access”
`XACE_SERVER_ACCESS`	XaceServerAccessRec	the section called “Server Access”
`XACE_SELECTION_ACCESS`	XaceSelectionAccessRec	the section called “Selection Access”
`XACE_SCREEN_ACCESS`	XaceScreenAccessRec	the section called “Screen Access”
`XACE_SCREENSAVER_ACCESS`	XaceScreenAccessRec	the section called “Screen Saver Access”
`XACE_AUTH_AVAIL`	XaceAuthAvailRec	the section called “Authorization Availability Hook”
`XACE_KEY_AVAIL`	XaceKeyAvailRec	the section called “Keypress Availability Hook”
`XACE_AUDIT_BEGIN`	XaceAuditRec	the section called “Auditing Hooks”
`XACE_AUDIT_END`	XaceAuditRec	the section called “Auditing Hooks”

In the descriptions that follow, it is helpful to have a listing of `Xext/xacestr.h` available for reference.

Core Dispatch

This hook allows security extensions to examine all incoming core protocol requests before they are dispatched. The hook argument is a pointer to a structure of type XaceCoreDispatchRec. This structure contains a *`client`* field of type ClientPtr and a *`status`* field of type int.

The *`client`* field refers to the client making the incoming request. Note that the complete request is accessible via the *`requestBuffer`* member of the client structure. The `REQUEST` family of macros, located in `include/dix.h`, are useful in verifying and reading from this member.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the request will not be processed further and the error code will be returned to the client.

Extension Dispatch

This hook allows security extensions to examine all incoming extension protocol requests before they are dispatched. The hook argument is a pointer to a structure of type XaceExtAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`ext`* field of type ExtensionEntry*, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client making the incoming request. Note that the complete request is accessible via the *`requestBuffer`* member of the client structure. The `REQUEST` family of macros, located in `include/dix.h`, are useful in verifying and reading from this member.

The *`ext`* field refers to the extension being accessed. This is required information since extensions are not associated with any particular major number.

The *`access_mode`* field is set to `DixUseAccess` when this hook is exercised.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the request will not be processed further and the error code will be returned to the client.

Resource Access

This hook allows security extensions to monitor all resource lookups. The hook argument is a pointer to a structure of type XaceResourceAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`id`* field of type XID, a *`rtype`* field of type RESTYPE, a *`res`* field of type pointer, a *`ptype`* field of type RESTYPE, a *`parent`* field of type pointer, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client on whose behalf the lookup is being performed. Note that this may be `serverClient` for server lookups.

The *`id`* field is the resource ID being looked up.

The *`rtype`* field is the type of the resource being looked up.

The *`res`* field is the resource itself: the result of the lookup.

The *`ptype`* field is the type of the parent resource or RT_NONE if not set.

The *`parent`* field is the parent resource itself or NULL if not set. The parent resource is set only when two conditions are met: The resource in question is being created at the time of the call (in which case the *`access_mode`* will include `DixCreateAccess`) and the resource in question has a defined parent object. Table 3, “Resource access hook parent objects." lists the resources that support parent objects. The purpose of these two fields is to provide generic support for "parent" resources.

The *access_mode* field encodes the type of action being performed. The valid mode bits are defined in `include/dixaccess.h`. The meaning of the bits depends on the specific resource type. Tables for some common types can be found in Table 2, "Resource access hook access modes.". Note that the `DixCreateAccess` access mode has special meaning: it signifies that the resource object is in the process of being created. This provides an opportunity for the security extension to initialize its security label information in the structure devPrivates or otherwise. If the status field is set to an error code in this case, the resource creation will fail and no entry will be made under the specified resource ID.

The *status* field may be set to a nonzero X protocol error code. In this event, the resource lookup will fail and an error (usually, but not always, the status value) will be returned to the client.

Table 2. Resource access hook access modes.

Access Mode Bit	Meaning	Example Call Site
`DixReadAccess`	The primary data or contents of the object are being read (drawables, cursors, colormaps).	GetImage, GetCursorImage, CreatePicture, QueryColors
`DixWriteAccess`	The primary data or contents of the object are being written (drawables, cursors, colormaps).	PutImage, RenderTriFan, ClearArea, StoreColors, RecolorCursor
`DixDestroyAccess`	The object is being removed.	CloseFont, DestroyWindow, FreePixmap, FreeCursor, RenderFreePicture
`DixCreateAccess`	The object is being created.	CreateWindow, CreatePixmap, CreateGC, CreateColormap
`DixGetAttrAccess`	The object's attributes are being queried, or the object is being referenced.	GetWindowAttributes, GetGeometry, QueryFont, CopyGC, QueryBestSize
`DixSetAttrAccess`	The object's attributes are being changed.	SetWindowAttributes, ChangeGC, SetClipRectangles, XFixesSetCursorName
`DixListPropAccess`	User properties set on the object are being listed (windows).	ListProperties
`DixGetPropAccess`	A user property set on the object is being read (windows).	GetProperty, RotateProperties
`DixSetPropAccess`	A user property set on the object is being written (windows).	ChangeProperty, RotateProperties, DeleteProperty

Access Mode Bit	**Meaning**	**Example Call Site**
`DixListAccess`	Child objects of the object are being listed out (windows).	QueryTree, MapSubwindows, UnmapSubwindows
`DixAddAccess`	A child object is being added to the object (drawables, fonts, colormaps).	CreateWindow, ReparentWindow, AllocColor, RenderCreatePicture, RenderAddGlyphs
`DixRemoveAccess`	A child object is being removed from object (drawables, fonts, colormaps).	DestroyWindow, ReparentWindow, FreeColors, RenderFreeGlyphs
`DixHideAccess`	Object is being unmapped or hidden from view (drawables, cursor).	UnmapWindow, XFixesHideCursor
`DixShowAccess`	Object is being mapped or shown (drawables, cursor).	MapWindow, XFixesShowCursor
`DixBlendAccess`	Drawable contents are being mixed in a way that may compromise contents.	Background "None", CompositeRedirectWindow, CompositeRedirectSubwindows
`DixGrabAccess`	Override-redirect bit on a window has been set.	CreateWindow, ChangeWindowAttributes
`DixInstallAccess`	Colormap is being installed.	InstallColormap
`DixUninstallAccess`	Colormap is being uninstalled.	UninstallColormap
`DixSendAccess`	An event is being sent to a window.	SendEvent
`DixReceiveAccess`	A client is setting an event mask on a window.	ChangeWindowAttributes, XiSelectExtensionEvent
`DixUseAccess`	The object is being used without modifying it (fonts, cursors, gc).	CreateWindow, FillPoly, GrabButton, ChangeGC
`DixManageAccess`	Window-manager type actions on a drawable.	CirculateWindow, ChangeSaveSet, ReparentWindow

Table 3. Resource access hook parent objects.

Resource Type	**Parent Resource Type**	**Notes**
`RT_WINDOW`	`RT_WINDOW`	Contains the parent window. This will be NULL for root windows.
`RT_PIXMAP`	`RT_WINDOW`	COMPOSITE extension only: the source window is passed as the parent for redirect pixmaps.

Resource Type	**Parent Resource Type**	**Notes**
`RenderPictureType`	`RC_DRAWABLE`	The source drawable is passed as the parent for Render picture objects.

Device Access

This hook allows security extensions to restrict client actions on input devices. The hook argument is a pointer to a structure of type XaceDeviceAccessRec. This structure contains a *client* field of type ClientPtr, a *dev* field of type DeviceIntPtr, a *access_mode* field of type Mask, and a *status* field of type int.

The *client* field refers to the client attempting to access the device (keyboard). Note that this may be `serverClient`.

The *dev* field refers to the input device being accessed.

The *access_mode* field encodes the type of action being performed. The valid mode bits are described in the table below.

The *status* field may be set to a nonzero X protocol error code. In this event, the device operation will fail and an error (usually, but not always, the status value) will be returned to the client.

Table 4. Device access hook access modes.

Access Mode Bit	**Meaning**	**Example Call Site**
`DixGetAttrAccess`	Attributes of the device are being queried.	GetKeyboardMapping, XiGetKeyboardControl, XkbGetDeviceInfo
`DixReadAccess`	The state of the device is being polled.	QueryPointer, QueryKeymap, XkbGetState
`DixWriteAccess`	The state of the device is being programatically manipulated.	WarpPointer, XTestFakeInput, XiSendExtensionEvent
`DixSetAttrAccess`	Per-client device configuration is being performed.	XkbPerClientFlags
`DixManageAccess`	Global device configuration is being performed.	ChangeKeyboardMapping, XiChangeDeviceControl, XkbSetControls
`DixUseAccess`	The device is being opened or referenced.	XiOpenDevice, XkbSelectEvents
`DixGrabAccess`	The device is being grabbed.	GrabPointer, GrabButton, GrabKey
`DixFreezeAccess`	The state of the device is being frozen by a synchronous grab.	GrabKeyboard, GrabPointer
`DixForceAccess`	The device cursor is being overriden by a grab.	GrabPointer, GrabButton

Access Mode Bit	Meaning	Example Call Site
`DixGetFocusAccess`	The device focus is being retrieved.	GetInputFocus, XiGetDeviceFocus
`DixSetFocusAccess`	The device focus is being set.	SetInputFocus, XiSetDeviceFocus
`DixBellAccess`	The device bell is being rung.	Bell, XiDeviceBell
`DixCreateAccess`	The device object has been newly allocated.	XIChangeDeviceHierarchy, XIAddMaster
`DixDestroyAccess`	The device is being removed.	XIChangeDeviceHierarchy, XIRemoveMaster
`DixAddAccess`	A slave device is being attached to the device.	XIChangeDeviceHierarchy, XIChangeAttachment
`DixRemoveAccess`	A slave device is being unattached from the device.	XIChangeDeviceHierarchy, XIChangeAttachment
`DixListPropAccess`	Properties set on the device are being listed.	ListDeviceProperties, XIListProperties
`DixGetPropAccess`	A property set on the device is being read.	GetDeviceProperty, XIGetProperty
`DixSetPropAccess`	A property set on the device is being written.	SetDeviceProperty, XISetProperty

Property Access

This hook allows security extensions to monitor all property accesses and additionally to support polyinstantiation if desired. The hook argument is a pointer to a structure of type XacePropertyAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`pWin`* field of type WindowPtr, a *`ppProp`* field of type PropertyPtr*, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client which is accessing the property. Note that this may be `serverClient` for server lookups.

The *`pWin`* field is the window on which the property is being accessed.

The *`ppProp`* field is a double-indirect pointer to the PropertyRec structure being accessed. The extra level of indirection supports property polyinstantiation; see below. If your extension does not use the polyinstantiation feature, simply dereference the pointer to obtain a PropertyPtr for the property

The *`access_mode`* field encodes the type of action being performed. The valid mode bits are described in the table below.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the property request will not be processed further and the error code will be returned to the client. However, the `BadMatch` code has special meaning; see below.

Table 5. Property access hook mode bits.

Access Mode Bit	Meaning	Example Call Site
`DixCreateAccess`	The property object has been newly allocated	ChangeProperty

Access Mode Bit	Meaning	Example Call Site
	(this bit will always occur in conjunction with `DixWriteAccess`).	
`DixWriteAccess`	The property data is being completely overwritten with new data.	ChangeProperty, RotateProperties
`DixBlendAccess`	The property data is being appended or prepended to.	ChangeProperty
`DixReadAccess`	The property data is being read.	GetProperty
`DixDestroyAccess`	The property data is being deleted.	DeleteProperty
`DixGetAttrAccess`	Existence of the property is being disclosed.	ListProperties
`DixPostAccess`	Post-write call reflecting new contents (this bit will always occur in conjunction with `DixWriteAccess`).	ChangeProperty

New in XACE Version 2.0, this hook supports the polyinstantiation of properties. This means that more than one property may exist having the same name, and the security extension can control which property object is seen by which client. To perform property polyinstantiation, your security extension should take the following steps:

- When a property is being created (`DixCreateAccess`), the security extension should label it appropriately based on the client that is creating it. In this case, the *`ppProp`* field should not be modified.
- When a property is being looked up, the *`ppProp`* field will refer to the first structure in the linked list with the given name. The security extension may change the ppProp field to a different property structure by traversing the linked list (using the PropertyRec *`next`* field) to find an alternate structure with the same property name.
- Alternately, when a property is being looked up, the *`status`* may be set to `BadMatch` which will cause the DIX layer to treat the property as not existing. This may result in an additional property object with the same name being created (in which case the hook will be called again with the create access mode).

New in XACE Version 2.2, this hook allows security extensions to verify the contents of properties after the client has written them. On a property change, the property access hook will be called twice. The first call is unchanged from previous versions. The second call will have the `DixPostAccess` bit together with `DixWriteAccess` and the *`ppProp`* property pointer will contain the new data. Setting the *`status`* field to something other than `Success` will cause the previous property contents to be restored and the client to receive the status code as an error.

Note that in the case of property creation (when `DixCreateAccess` is set), the *`ppProp`* field already reflects the new data. Hence security extensions wishing to validate property data should check for either `DixPostAccess` or `DixCreateAccess` in conjunction with `DixWriteAccess`. If your extension does not need this feature, simply ignore calls with the `DixPostAccess` bit set.

Send Access

This hook allows security extensions to prevent devices and clients from posting X events to a given window. The hook argument is a pointer to a structure of type XaceSendAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`dev`* field of type DeviceIntPtr, a *`pWin`* field of type WindowPtr, a *`events`* field of type events, a *`count`* field of type int, and a *`status`* field of type int.

The *`client`* field refers to the client attempting a `SendEvent` request or other synthetic event generation to the given window. This field may be NULL if the *`dev`* field is set.

The *`dev`* field refers to the device attempting to post an event which would be delivered to the given window. This field may be NULL if the *`client`* field is set.

The *`pWin`* field refers to the target window.

The *`events`* field refers to the events that are being sent.

The *`count`* field contains the number of events in the *`events`* array.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the events will be dropped on the floor instead of being delivered.

Warning

This hook does not currently cover all instances of event delivery.

Receive Access

This hook allows security extensions to prevent a client from receiving X events that have been delivered to a given window. The hook argument is a pointer to a structure of type XaceReceiveAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`pWin`* field of type WindowPtr, a *`events`* field of type events, a *`count`* field of type int, and a *`status`* field of type int.

The *`client`* field refers to the client to which the event would be delivered.

The *`pWin`* field refers to the window where the event has been sent.

The *`events`* field refers to the events that are being sent.

The *`count`* field contains the number of events in the *`events`* array.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the events will not be delivered to the client.

Warning

This hook does not currently cover all instances of event delivery.

Client Access

This hook allows security extensions to prevent clients from manipulating other clients directly. This hook applies to a small set of protocol requests such as `Kill-`

`Client`. The hook argument is a pointer to a structure of type XaceClientAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`target`* field of type ClientPtr, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client making the request.

The *`target`* field refers to the client being manipulated.

The *`access_mode`* field encodes the type of action being performed. The valid mode bits are described in the table below.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the request will fail and an error (usually, but not always, the status value) will be returned to the client.

Table 6. Client access hook mode bits.

Access Mode Bit	**Meaning**	**Example Call Site**
`DixGetAttrAccess`	Attributes of the client are being queried.	SyncGetPriority
`DixSetAttrAccess`	Attributes of the client are being set.	SyncSetPriority
`DixManageAccess`	The client's close-down-mode (which affects global server resource management) is being set.	SetCloseDownMode
`DixDestroyAccess`	The client is being killed.	KillClient

Extension Access

This hook allows security extensions to approve or deny requests involving which extensions are supported by the server. This allows control over which extensions are visible. The hook argument is a pointer to a structure of type XaceExtAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`ext`* field of type ExtensionEntry*, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client making the incoming request, which is typically `QueryExtension` or `ListExtensions`.

The *`ext`* field refers to the extension being accessed. This is required information since extensions are not associated with any particular major number.

The *`access_mode`* field is set to `DixGetAttrAccess` when this hook is exercised.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the extension will be reported as not supported (`QueryExtensions`) or omitted from the returned list (`ListExtensions`).

Warning

If this hook is used, an extension dispatch hook should also be installed to make sure that clients cannot circumvent the check by guessing the major opcodes of extensions.

Server Access

This hook allows security extensions to approve or deny requests that affect the X server itself. The hook argument is a pointer to a structure of type XaceServerAccessRec, which contains a *`client`* field of type ClientPtr, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client making the request.

The *`access_mode`* field encodes the type of action being performed. The valid mode bits are described in the table below.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the request will fail and an error (usually, but not always, the status value) will be returned to the client.

Table 7. Server access hook mode bits.

Access Mode Bit	Meaning	Example Call Site
`DixGetAttrAccess`	Attributes of the server are being queried.	GetFontPath
`DixSetAttrAccess`	Attributes of the server are being set.	SetFontPath
`DixManageAccess`	Server management is being performed.	ChangeAccessControl, ListHosts
`DixGrabAccess`	A server grab is being performed.	GrabServer
`DixReadAccess`	The server's actions are being recorded.	Record, XEVIE extensions
`DixDebugAccess`	Server debug facilities are being used.	XTest extension, XkbSetDebuggingFlags

Selection Access

This hook allows security extensions to monitor all selection accesses and additionally to support polyinstantiation if desired. The hook argument is a pointer to a structure of type XaceSelectionAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`ppSel`* field of type Selection**, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client which is accessing the property. Note that this may be `serverClient` for server lookups.

The *`ppSel`* field is a double-indirect pointer to the Selection structure being accessed. The extra level of indirection supports selection polyinstantiation; see below. If your extension does not use the polyinstantiation feature, simply dereference the pointer to obtain a SelectionRec * for the selection.

The *`access_mode`* field encodes the type of action being performed. The valid mode bits are described in the table below.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the property request will not be processed further and the error code will be returned to the client. However, the `BadMatch` code has special meaning; see below.

Table 8. Selection access hook mode bits.

Access Mode Bit	**Meaning**	**Example Call Site**
`DixCreateAccess`	The selection object has been newly allocated (this bit will always occur in conjunction with `DixSetAttrAccess`).	SetSelectionOwner
`DixSetAttrAccess`	The selection owner is being set.	SetSelectionOwner
`DixGetAttrAccess`	The selection owner is being queried.	GetSelectionOwner
`DixReadAccess`	A convert operation is being requested on the selection.	ConvertSelection

This hook supports the polyinstantiation of selections. This means that more than one selection may exist having the same name, and the security extension can control which selection object is seen by which client. To perform selection polyinstantiation, your security extension should take the following steps:

- When selection ownership is being established (`DixSetAttrAccess`), the security extension should label it appropriately based on the client that is taking ownership. In this case, the *`ppSel`* field should not be modified.

- When a selection is being looked up, the *`ppProp`* field will refer to the first structure in the linked list with the given name. The security extension may change the ppSel field to a different selection structure by traversing the linked list (using the Selection *`next`* field) to find an alternate structure with the same selection name.

- Alternately, when a selection is being looked up, the *`status`* may be set to `BadMatch` which will cause the DIX layer to treat the selection as not existing. This may result in an additional selection object with the same name being created (in which case the hook will be called again with the create access mode).

Screen Access

This hook allows security extensions to approve or deny requests that manipulate screen objects The hook argument is a pointer to a structure of type XaceScreenAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`screen`* field of type ScreenPtr, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client making the request.

The *`screen`* field refers to the screen object being referenced.

The *`access_mode`* field encodes the type of action being performed. The valid mode bits are described in the table below.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the request will not be processed further and the error code will be returned to the client.

Table 9. Screen access hook mode bits.

Access Mode Bit	Meaning	Example Call Site
`DixGetAttrAccess`	Attributes of the screen object are being queried.	ListInstalledColormaps, QueryBestSize
`DixSetAttrAccess`	Attributes of the screen object are being set.	InstallColormap
`DixHideAccess`	The cursor on the screen is being globally hidden.	XFixesHideCursor
`DixShowAccess`	The cursor on the screen is being globally unhidden.	XFixesShowCursor

Screen Saver Access

This hook allows security extensions to approve or deny requests that manipulate the screensaver. The hook argument is a pointer to a structure of type XaceScreenAccessRec. This structure contains a *`client`* field of type ClientPtr, a *`screen`* field of type ScreenPtr, a *`access_mode`* field of type Mask, and a *`status`* field of type int.

The *`client`* field refers to the client making the request.

The *`screen`* field refers to the screen object being referenced.

The *`access_mode`* field encodes the type of action being performed. The valid mode bits are described in the table below.

The *`status`* field may be set to a nonzero X protocol error code. In this event, the request will not be processed further and the error code will be returned to the client.

Table 10. Screen saver access hook mode bits.

Access Mode Bit	Meaning	Example Call Site
`DixGetAttrAccess`	Attributes of the screen saver are being queried.	GetScreenSaver, ScreenSaverQueryInfo
`DixSetAttrAccess`	Attributes of the screen saver are being set.	SetScreenSaver, ScreenSaverSelectInput
`DixHideAccess`	The screen saver is being programmatically activated.	ForceScreenSaver, DPMSEnable
`DixShowAccess`	The screen saver is being programmatically deactivated.	ForceScreenSaver, DPMSDisable

Authorization Availability Hook

This hook allows security extensions to examine the authorization associated with a newly connected client. This can be used to set up client security state depending on the authorization method that was used. The hook argument is a pointer to a structure of type XaceAuthAvailRec. This structure contains a *`client`* field of type ClientPtr, and a *`authId`* field of type XID.

The *`client`* field refers to the newly connected client.

The *`authId`* field is the resource ID of the client's authorization.

This hook has no return value.

Note

This hook is called after the client enters the initial state and before the client enters the running state. Keep this in mind if your security extension uses the `ClientStateCallback` list to keep track of clients.

This hook is a legacy of the APPGROUP Extension. In the future, this hook may be phased out in favor of a new client state, `ClientStateAuthenticated`.

Keypress Availability Hook

This hook allows security extensions to examine keypresses outside of the normal event mechanism. This could be used to implement server-side hotkey support. The hook argument is a pointer to a structure of type XaceKeyAvailRec. This structure contains a *`event`* field of type xEventPtr, a *`keybd`* field of type DeviceIntPtr, and a *`count`* field of type int.

The *`event`* field refers to the keyboard event, typically a `KeyPress` or `KeyRelease`.

The *`keybd`* field refers to the input device that generated the event.

The *`count`* field is the number of repetitions of the event (not 100% sure of this at present, however).

This hook has no return value.

Auditing Hooks

Two hooks provide basic auditing support. The begin hook is called immediately before an incoming client request is dispatched and before the dispatch hook is called (refer to the section called “Core Dispatch”). The end hook is called immedately after the processing of the request has finished. The hook argument is a pointer to a structure of type XaceKeyAvailRec. This structure contains a *`client`* field of type ClientPtr, and a *`requestResult`* field of type int.

The *`client`* field refers to client making the request.

The *`requestResult`* field contains the result of the request, either `Success` or one of the protocol error codes. Note that this field is significant only in the end hook.

These hooks have no return value.

Protocol

Requests

XACE does not define any X protocol.

Events

XACE does not define any X protocol.

Errors

XACE does not define any X protocol.