

# **X Access Control Extension Specification**

**Eamon F. Walsh**  
2006

## **Revision History**

Revision 1.0 19 Oct 2006 Revised by: efw  
Initial Version

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 is a generalization of the "Security" extension, which provides a simple on/off trust model, with untrusted windows being restricted in certain areas. Most of XACE consists simply of replacing the Security-specific checks in the dix/os layer with generic callback lists. However, the framework is flexible enough to allow for hooks to be added or deprecated in the future.

This paper describes the implementation of XACE, 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.

## **1. Introduction**

### **1.1. 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. It is highly recommended that the reader review the specifications for the SECURITY and APPGROUP extensions. The relevant documents are all available in the X.Org documentation package.

## 1.2. 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.

## 1.3. Prior Work

### 1.3.1. Security Extension

XACE is primarily 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.

The primary limitation of the SECURITY extension is a lack of granularity. The trust level of client connections is set at connection time and is not changed. Creating untrusted clients is cumbersome since untrusted authorizations must be generated dynamically (they cannot be specified in the authorization file used by the server at startup time), and the default trusted behavior is not restricted in any way. The author of the SECURITY extension did anticipate the need for flexibility in some areas, but the XACE modifications introduce a much broader level of generalization.

The benefit of the SECURITY extension is that its authors already had identified the proper places in the core X server code to place their trust level checks. Thus, the only thing that needed to be done for XACE was to swap out these checks for more generic XACE hooks. The SECURITY authors also introduced several features into the core server code, such as the "Security" versions of the resource lookup functions, and the custom ProcVectors used to dispatch client requests.

With the introduction of XACE, the SECURITY extension has been rewritten to sit on top of XACE, thus maintaining backwards compatibility.

### **1.3.2. 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.

### **1.3.3. Linux Security Modules**

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

## **1.4. Future Work**

### **1.4.1. Security Hooks**

It is anticipated that the set of security hooks provided by XACE will change with time. Some of the current hooks provide legacy functionality for the SECURITY extension and 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 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 block the request before it is handled by the server (resulting in a `BadAccess` error). 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, as the `SECURITY` extension has done.

In the future, it may be worthwhile to integrate XACE directly into the X server code, removing its status as an "extension" (XACE defines no new protocol). This would eliminate the `ifdef` directives that bracket the XACE hooks, and would allow for further integration with the surrounding code. It would also avoid the need to use the extension loader to initialize XACE. The use of modern coding techniques such as static inlining could also be used to improve performance in the hook mechanism.

### 1.4.2. X Authentication

The X server supports several authentication methods. Currently, they are implemented directly in the OS layer of the X server. However, with new improvements to the Pluggable Authentication Modules (PAM) library, it may be possible to move these authentication methods out of the server, implementing each one as a PAM module. This would separate security-specific code from the X server, as well allow the authentication code in the OS layer to be cleaned up significantly. However, the author has not studied the problem in great detail, so it's too early to tell whether this idea is workable.

Another area where the X authentication code could use some cleanup is the `SECURITY` extension's "Query Security" authentication pseudo-method. This method is used to determine whether or not an X server supports certain "site policies," identified via strings. This method can also be used to assert requirements about extension security. This part of the `SECURITY` extension was not refactored along with the rest of the extension as part of the XACE work. As part of the PAM project described above, it would be beneficial if this authentication method could be moved out to a PAM module or simply dropped. Doing this would allow the `SECURITY` extension to be loaded as a module instead of being built-in, since the auth code is the only remaining part of that extension that needs to be compiled in.

### 1.4.3. Core X Server

There are some minor improvements that could be made to the core X server itself outside of XACE. As will be discussed, security extension writers are expected to use the `devPrivates` mechanism to store security state with various server object. This mechanism is currently duplicated for each structure type that supports it; it may be possible to use macros to generate the functions for each supported structure type, at least reducing the code size. This would also make it easier to support more structures; only a new structure field would be required along with slight changes to the code that allocates the structure. Extending `devPrivates` support to other structures, or even generic resources, would be beneficial for security extension writers. The feasibility of doing this generalization is currently being investigated by the author. In addition, initialization and teardown callbacks are needed as described in Section 2.1.2; support for them is currently spotty.

The module loader should be looked at to see if the extension loading sequence could be improved. There are comments to that effect in the module loading code, which read (paraphrasing): "Please make extension loading not suck." Right now, there are two initialization functions that extensions can use: a "setup" function which is called first, before any `ExtensionEntry` structures are created, and an "init" function which is called when the structure is created. This is OK, but the order in which the setup functions are called is odd: loadable extensions are called first, before built-in extensions. The calls also happen from totally different places in the code, with loadables being set up from the `InitOutput` function which is nonintuitive. Finally, the extension support code has large, cumbersome lists of extensions bracketed by `ifdef`'s, along with boolean variables meant to be used for dynamic configuration of extensions which are in practice unused. Perhaps autotools could be used to build the list of extensions to load, instead of having a hard-coded list. The author is investigating possibilities for work in this area.

## 2. Usage

### 2.1. 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. Two methods of storing security state are discussed here.

#### 2.1.1. Global Variables

One method of storing state is simply to use global variables in the extension code. Tables can be kept corresponding to internal server structures, updated to stay synchronized with the structures themselves.

#### 2.1.2. Device Privates

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. However, only the server structures listed in Table 1 currently support this mechanism; work is in progress to add other structure types (see Section 1.4.3).

**Table 1. Current `devPrivates` support in DIX.**

Structure	Supports Pre-Allocation	Cleared to Zero	Callbacks Available
ClientRec	Yes	Yes	Init/Free
ExtensionEntry	Yes	Yes	No
ScreenRec	No	Yes	No

Structure	Supports Pre-Allocation	Cleared to Zero	Callbacks Available
WindowRec	Yes	No	Init
GCRec	Yes	No	No
PixmapRec	Yes	No	No
ColormapRec	No	Yes	Init
DeviceIntRec	No	Yes	No

For an example of how to use `devPrivates`, refer to the SECURITY extension source code in `Xext/security.c` which makes use of them for storing state in the `ClientRec` and `ExtensionEntry` structures. Basically, your extension must register for space in each structure type. This is done slightly differently depending on the structure; see the SECURITY example as well as `dix/privates.c`, which contains the implementation. All structures provide an instance of `DevUnion`, indexed by a number that is returned to you in the `devPrivates` array member of the structure. This union can be used as a long value or a pointer. Some structures allow a byte count to be provided at registration time, which will be automatically allocated and returned through the pointer member of the union.

The registration must be done in the extension setup routine for the `ExtensionEntry` structure only; for all other structures it can be performed in the extension init routine. See the SECURITY code for registration examples.

When a structure having `devPrivates` support is allocated, the space requested by all registrants is allocated along with it. In some cases, the newly allocated memory is cleared to zero. Work is underway to make sure that all supported structures have the memory cleared; the ones that currently do are listed in Table 1. However, your security extension may need to take further action to initialize the newly created data. How exactly this is done depends on the structure. Again, some structures don't currently provide a way for your extension to be called immediately when a structure instance is created. The `ClientRec` and `WindowRec` structures do have support for this, as indicated in Table 1. Examine the SECURITY source for more information. The eventual goal is to have a callback, XACE or otherwise, notifying when each supported structure is initialized.

The same applies to freeing memory or otherwise tearing down your security state when an object is being destroyed. Some structures don't currently have callbacks associated with this event which would allow a security extension to gain control. The `ClientRec` structure does have support. The eventual goal is to provide a mechanism for this purpose.

**Note:** Memory allocated through the `devPrivates` mechanism itself will be freed automatically.

## 2.2. Using Hooks

### 2.2.1. 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. 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 callback 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 2 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.

## 2.2.2. Hooks

The currently defined set of XACE hooks is shown in Table 2. As discussed in Section 1.4.1, 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 2. XACE security hooks.**

Hook Identifier	Callback Argument Type	Refer to
XACE_CORE_DISPATCH	XaceCoreDispatchRec	Section 2.2.2.1
XACE_EXT_DISPATCH	XaceExtAccessRec	Section 2.2.2.2
XACE_RESOURCE_ACCESS	XaceResourceAccessRec	Section 2.2.2.3
XACE_PROPERTY_ACCESS	XacePropertyAccessRec	Section 2.2.2.4
XACE_MAP_ACCESS	XaceMapAccessRec	Section 2.2.2.5
XACE_DRAWABLE_ACCESS	XaceDrawableAccessRec	Section 2.2.2.6
XACE_BACKGRND_ACCESS	XaceMapAccessRec	Section 2.2.2.7
XACE_DEVICE_ACCESS	XaceDeviceAccessRec	Section 2.2.2.8
XACE_HOSTLIST_ACCESS	XaceHostlistAccessRec	Section 2.2.2.9
XACE_EXT_ACCESS	XaceExtAccessRec	Section 2.2.2.10
XACE_WINDOW_INIT	XaceWindowRec	Section 2.2.2.11
XACE_AUTH_AVAIL	XaceAuthAvailRec	Section 2.2.2.12
XACE_KEY_AVAIL	XaceKeyAvailRec	Section 2.2.2.13
XACE_AUDIT_BEGIN	XaceAuditRec	Section 2.2.2.14
XACE_AUDIT_END	XaceAuditRec	Section 2.2.2.14

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

### 2.2.2.1. 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 *rval* 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 *rval* field should be set to `FALSE` if the request is to be denied. The result of a denied request is a `BadAccess` error, which is delivered to the client.



### 2.2.2.2. 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*`, and a `rval` 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 `rval` field should be set to `FALSE` if the request is to be denied. The result of a denied request is a `BadAccess` error, which is delivered to the client.

### 2.2.2.3. 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 `access_mode` field of type `Mask`, a `res` field of type pointer, and a `rval` 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 resource type being looked up.

The `access_mode` field encodes the type of action being performed. The valid values are defined in `include/resource.h` (look for `SecurityReadAccess`). This field is a legacy of the `SECURITY` extension.

## Warning

The `access_mode` field is not widely used by the core server and is often the default "unknown" value. The semantics of this field may be changed in the future.

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

The *rval* field should be set to `FALSE` if the lookup is to be denied. The result of a denied request is a lookup failure, which will have varying effects on the client (or server) depending on the type of resource.

#### 2.2.2.4. Property Access

This hook allows security extensions to monitor all property accesses. 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 *propertyName* field of type `Atom`, a *access\_mode* field of type `Mask`, and a *rval* 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 *propertyName* field is the name of the property being accessed.

The *access\_mode* field encodes the type of action being performed. The valid values are defined in `include/resource.h` (look for `SecurityReadAccess`). This field is a legacy of the SECURITY extension.

The *rval* field should be set to one of the Operation constants defined in `Xext/xace.h`. The options are to allow, deny, or ignore the request. The difference between denying and ignoring is that an ignored request returns successfully but either does nothing (for a write) or returns an empty string (for a read).

### Warning

The semantics of the *access\_mode* field and the *rval* field may be changed in the future. See the warning in Section 2.2.2.3.

#### 2.2.2.5. Map Access

This hook allows security extensions to approve or deny requests to map windows. The hook argument is a pointer to a structure of type `XaceMapAccessRec`. This structure contains a *client* field of type `ClientPtr`, a *pWin* field of type `WindowPtr`, and a *rval* field of type `int`.

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

The *pWin* field refers to the window being mapped.

The *rval* field should be set to `FALSE` if the request is to be denied. Currently, the result of a denied request is a successful return leaving the window unmapped. In the future, this may be changed to return a `BadAccess` error to the client.

#### 2.2.2.6. Drawable Access

This hook allows security extensions to force censoring of overlapping windows when a `GetImage` request is made. Refer to the "Image Security" section of the SECURITY extension specification for more information. The hook argument is a pointer to a structure of type `XaceDrawableAccessRec`. This structure contains a *client* field of type `ClientPtr`, a *pDraw* field of type `DrawablePtr`, and a *rval* field of type `int`.

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

The *pDrawable* field refers to the subject of the `GetImage` request.

The *rval* field should be set to `FALSE` if the drawable is to be checked for overlapping windows and censored appropriately.

#### 2.2.2.7. Background Access

This hook allows security extensions to force censoring of background "None" windows, preventing contents of other windows from showing through. The hook argument is a pointer to a structure of type `XaceMapAccessRec`. This structure contains a *client* field of type `ClientPtr`, a *pWin* field of type `WindowPtr`, and a *rval* field of type `int`.

The *client* field refers to the client making the request, typically a `CreateWindow` request.

The *pWin* field refers to the window being created.

The *rval* field should be set to `FALSE` if the background is to be censored.

### Warning

This hook may be merged with the drawable access hook at some point in the future.

#### 2.2.2.8. Device Access

This hook allows security extensions to restrict certain actions by clients related to keyboard input. For the specifics, refer to the "Input Security" section of the SECURITY extension specification. 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 *fromRequest* field of type `Bool`, and a *rval* 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 *fromRequest* field is `TRUE` if the access is the result of a client request; `FALSE` otherwise.

The *rval* field should be set to `FALSE` if the client is to be restricted. The result of the return value varies depending on the context of the call.

### Warning

This hook is a legacy of the SECURITY extension and covers only the core server. Extensions do exist, such as XEVIE, that allow clients to intercept and modify input events.

The input subsystem in X.Org is in a state of change and it is expected that input event security will not be fully addressed until later versions of XACE.

#### 2.2.2.9. Host List Access

This hook allows security extensions to approve or deny requests to read or change the host access list. The hook argument is a pointer to a structure of type `XaceHostlistAccessRec`. This structure contains a *client* field of type `ClientPtr`, a *access\_mode* field of type `Mask`, and a *rval* 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 values are defined in `include/resource.h` (look for `SecurityReadAccess`). Currently this field is set to read access for `ProcListHosts` and write access otherwise.

The *rval* field should be set to `FALSE` if the request is to be denied. The result of a denied request is a `BadAccess` error, which is delivered to the client.

#### 2.2.2.10. Extension Access

This hook allows security extensions to approve or deny requests involving supported server extensions. 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*`, and a *rval* field of type `int`.

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

The *ext* field refers to the extension being queried.

The *rval* field should be set to `FALSE` if the client is not to be made aware of the extension. For a `QueryExtension` request, a denial results in a response indicating the extension is not present. For a `ListExtensions` request, a denial results in the exclusion of the extension from the returned list.

#### 2.2.2.11. Window Initialization

This hook allows security extensions to set up security state for newly created windows. The hook argument is a pointer to a structure of type `XaceWindowRec`. This structure contains a *client* field of type `ClientPtr`, and a *pWin* field of type `WindowPtr`.

The *client* field refers to the client owning the window. Note that this may be `serverClient`.

The *pWin* field refers to the newly created window.

This hook has no return value.

#### 2.2.2.12. 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`.

### 2.2.2.13. 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.

## Warning

The warning in Section 2.2.2.8 applies to this hook. This hook is provided mainly for support of the Trusted Solaris extension.

### 2.2.2.14. 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 Section 2.2.2.1). The end hook is called immediately 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.

## **3. Protocol**

### **3.1. Requests**

XACE does not define any X protocol.

### **3.2. Events**

XACE does not define any X protocol.

### **3.3. Errors**

XACE does not define any X protocol.