Analysis of the X Protocol for Security Concerns

Draft Version 2

David Wiggins

Analysis of the X Protocol for Security Concerns: Draft Version 2

by David Wiggins

conversion from tex to docbook: Matt Dew

Abstract

This paper attempts to list all instances of certain types of security problems in the X Protocol. Issues with authorization are not addressed. We assume that a malicious client has already succeeded in connecting, and try to assess what harm it can then do. We propose modifications to the semantics of the X Protocol to reduce these risks.

Table of Contents

Chapter 1. Definition of Threats

We analyze the X protocol for the following threats.

Theft	occurs when a client gains access to information owned by another client without explicit permission from that other client. For this analysis, we take a broad view of ownership: any information that exists in the server due to the actions of a client is considered owned by that client. Furthermore, the client that has input focus owns keyboard events, and the client that owns the window that the pointer is in owns mouse events. This view may reveal certain instances of "theft" that we don't care to stop, but we think it is better to identify all potential candidates up front and cull the list later than to do a partial analysis now and plan on reanalyzing for remaining holes later.
Denial of service	occurs when a client causes another client or the user to lose the ability to perform some operation.
Spoofing	occurs when a client attempts to mimic another client with the hope that the user will interact with it as if it really were the mimicked client. A wide variety of requests may be used in a spoofing attack; we will only point out a few likely candidates.
Destruction	occurs when a client causes another client to lose information in a way that the client or user is likely to notice. (This does not count expected forms of destruction, e.g., exposures.)
Alteration	occurs when a client causes another client to lose information in a way that the client or user is unlikely to notice. e.g., changing one pixel in a drawable.

The line between alteration and destruction is subjective. Security literature does often distinguish between them, though not always explicitly. Alteration is often considered more insidious because its effects may not be realized until long after it has occurred. In the intervening time, each time the altered data is used, it can cause more damage.

Chapter 2. General security concerns and remedies

The following sections discuss security problems intrinsic to the X Protocol. A statement of each problem is usually followed by potential remedies. A few words here about possible remedies will help frame the specific ones described below.

If a client attempts a threatening operation, the server may take one of the following actions, listed roughly in order of severity:

1. Execute the request normally. This is the right choice when we decide that a particlar threat is not serious enough to worry about.

2. Execute the request in some modified form, e.g., substitute different values for some of the request fields, or edit the reply.

3. Arrange to ask the user what to do, given some subset of the other choices in this list. This must be used sparingly because of the performance impact.

4. Treat the request as a no-op. If the client will probably not notice, or if it seems likely that the intent was benign, this is a good choice.

5. Send a protocol error to the client. If the client will be confused enough by the other options that it will probably crash or become useless anyway, or if it seems likely that the intent was malicious, this is a good choice.

   Kill the client. This might be the right action if there is no doubt that the client is hostile.

In most cases we present the one option that seems most appropriate to counter the threat, taking into account the seriousness of the threat, the implementation difficulty, and the impact on applications. Our initial bias is to err on the side of stronger security, with the accompanying tighter restrictions. As we uncover important operations and applications that the new restrictions interfere with, we can apply selective loosening to allow the desired functionality.

In some cases we will suggest returning an Access error where the X protocol does not explicitly allow one. These new Access errors arise when a client can only perform a (non-empty) subset of the defined operations on a resource. The disallowed operations cause Access errors. The resiource at issue is usually a root window.

Access to Server Resources

The X protocol allows clients to manipulate resources (objects) belonging to other clients or to the server. Any request that specifies a resource ID is vulnerable to some of the above threats. Such requests also provide a way for a client to guess resource IDs of other clients. A client can make educated guesses for possible resource IDs, and if the request succeeds, it knows it has determined a valid resource ID. We call this "resource ID guessing" in the list below.

One likely defense against these problems is to have the server send an appropriate protocol error to deny the existence of any resource specified by a client that doesn't belong to that client. A variation on this policy lets cooperating groups of clients access each other's resources, but not those of other groups. The Broadway project will initially use a less general form of this idea by having two groups, trusted and untrusted. Trusted clients can do everything that X clients can do today. They will be protected from untrusted clients in ways described below. Untrusted clients will not be protected from each other. Though this will be the initial design, we need to make sure there is a growth path to multiple (more than two) groups.

Most of the time, applications never access server resources that aren't their own, so the impact of disallowing such accesses should be minimal. There are a few notable exceptions, most of which will be discussed under the relevant protocol requests. They are: ICCCM selection transfer, Motif drag and drop, and server-global resources like the root window and default colormap. Another major exception is the window manager, which routinely manipulates windows of other applications. The solution for window managers is to always run them as trusted applications.

The implementation difficulty of limiting access to resources should not be large. All resource accesses eventually funnel down to one of two functions in <dix/resource.c>: `LookupIDByType` and `LookupIDByClass`. A few lines of checking at the top of these functions will form the heart of this defense. There is a small problem because these functions are not told which client is doing the lookup, but that can be solved either by adding a client parameter (probably as a new function to preserve compatibility), or by using the server global requestingClient.

Note

ISSUE: are we really going to be able to get away with hiding trusted resources, or will things like Motif drag and drop force us to expose them? (Either way, the operations that untrusted clients can do to trusted resources will have to be limited.) Is there something in Xt or the ICCCM that breaks if you hide resources?

Denial of Service

Memory Exhaustion

Any request that causes the server to consume resources (particularly memory) can be used in a denial of service attack. A client can use such requests repeatedly until the server runs out of memory. When that happens, the server will either crash or be forced to send Alloc errors. The most obvious candidates are resource creation requests, e.g., CreatePixmap, but in reality a large percentage of requests cause memory allocation, if only temporarily, depending on the server implementation. For this reason, the list of requests subject to this form of denial of service will be necessarily incomplete.

To address this form of denial of service, the server could set per-client quotas on memory consumption. When the limit is surpassed, the server could return Alloc errors. The application impact is minimal as long as the application stays within quota. The implementation difficulty is another story.

Conceptually, it seems easy: simply have a way to set the limit, and on every memory (de)allocation operation, update the client's current usage, and return an error if the client is over the limit. The first problem is something we've already touched on: the allocator functions aren't told which client the allocation belongs to. Unlike resource lookups, allocations are done in too many places to consider a new interface that passes the client, so using the global requestingClient is practically mandatory.

The problems run deeper. The logical thing for the allocator to do if the client is over its limit is to return NULL, indicating allocation failure. Unfortunately, there are many places in the server that will react badly if this happens. Most of these places, but not all, are "protected" by setting the global variable Must_have_memory to True around the delicate code. We could help the problem by skipping the limit check if Must_have_memory is True. The best solution would be to bullet-proof the server against allocation failures, but that is beyond the scope of Broadway. Another consideration is that the additional checking may have a measurable performance impact, since the server does frequent allocations.

A third problem is that there is no portable way to determine the size of a chunk of allocated memory given just a pointer to the chunk, and that's all you have inside `Xrealloc` and `Xfree`. The server could compensate by recording the sizes itself somewhere, but that would be wasteful of memory, since the malloc implementation also must be recording block sizes. On top of that, the redundant bookkeeping would hurt performance. One solution is to use a custom malloc that has the needed support, but that too seems beyond the scope of Broadway.

Considering all of this, we think it is advisable to defer solving the memory exhaustion problem to a future release. Keep this in mind when you see quotas mentioned as a defense in the list below.

CPU Monopolization

Another general way that a client can cause denial of service is to flood the server with requests. The server will spend a large percentage of its time servicing those requests, possibly starving other clients and certainly hurting performance. Every request can be used for flooding, so we will not bother to list flooding on every request. A variation on this attack is to flood the server with new connection attempts.

To reduce the effectiveness of flooding, the server could use a different scheduling algorithm that throttles clients that are monopolizing the server, or it could simply favor trusted clients over untrusted ones. Applications cannot depend on a particular scheduling algorithm anyway, so changing it should not affect them. The Synchronization extension specifies a way to set client priorities, and a simple priority scheduler already exists in the server to support it, so this should be simple to add.

Chapter 3. Security concerns with specific window attributes

Background-pixmap

Clients can use windows with the background-pixmap attribute set to None (hereafter "background none windows") to obtain images of other windows. A background none window never paints its own background, so whatever happened to be on the screen when the window was mapped can be read from the background none window with GetImage. This may well contain data from other windows. The CreateWindow and ChangeWindowAttributes requests can set the background-pixmap attribute set to None, and many window operations can cause data from other windows to be left in a background none window, including `ReparentWindow`, `MapWindow`, `MapSubwindows`, `ConfigureWindow`, and `CirculateWindow`.

Background none windows can also be used to cause apparent alteration. A client can create a window with background none and draw to it. The drawing will appear to the user to be in the windows below the background none window.

To remedy these problems, the server could substitute a well-defined background when a client specifies None. Ideally the substituted background would look different enough from other windows that the user wouldn't be confused. A tile depicting some appropriate international symbol might be reasonable. We believe that there are few applications that actually rely on background none semantics, and those that do will be easy for the user to identify because of the distinctive tile. Implementation should not be a problem either. Luckily, the window background cannot be retrieved through the X protocol, so we won't have to maintain any illusions about its value.

Note

ISSUE: Some vendors have extensions to let you query the window background. Do we need to accomodate that?

Note

ISSUE: Will this lead to unacceptable application breakage? Could the server be smarter, only painting with the well-defined background when the window actually contains bits from trusted windows?

ParentRelative and CopyFromParent

Several window attributes can take on special values that cause them to reference (ParentRelative) or copy (CopyFromParent) the same attribute from the window's parent. This fits our definition of theft. The window attributes are class, background-pixmap, border-pixmap, and colormap. All of these can be set with `CreateWindow`; all but class can be set with `ChangeWindowAttributes`.

These forms of theft aren't particularly serious, so sending an error doesn't seem appropriate. Substitution of different attribute values seems to be the only reasonable option, and even that is likely to cause trouble for clients. Untrusted clients are already going to be prevented from creating windows that are children of trusted clients (see CreateWindow below). We recommend that nothing more be done to counter this threat.

Override-redirect

Windows with the override-redirect bit set to True are generally ignored by the window manager. A client can map an override-redirect window that covers most or all of the screen, causing denial of service since other applications won't be visible.

To prevent this, the server could prevent more than a certain percentage (configurable) the of screen area from being covered by override-redirect windows of untrusted clients.

Override-redirect windows also make some spoofing attacks easier since the client can more carefully control the presentation of the window to mimic another client. Defenses against spoofing will be given under the section called “MapWindow” .

Chapter 4. Security concerns with specific requests

To reduce the space needed to discuss 120 requests, most of the following sections use a stylized format. A threat is given, followed by an imperative statement. The implied subject is an untrusted client, and the object is usually a trusted client. Following that, another statement starting with "Defense:" recommends a countermeasure for the preceding threat(s).

Resources owned by the server, such as the root window and the default colormap, are considered to be owned by a trusted client.

CreateWindow

Alteration: create a window as a child of another client's window, altering its list of children.

Defense: send Window error. Specifying the root window as the parent will have to be allowed, though.

Theft: create an InputOnly window or a window with background none on top of other clients' windows, select for keyboard/mouse input on that window, and steal the input. The input can be resent using SendEvent or an input synthesis extension so that the snooped application continues to function, though this won't work convincingly with the background none case because the drawing will be clipped.

Defense: send an error if a top-level InputOnly window is created (or reparented to the root). Countermeasures for background none and SendEvent are discussed elsewhere.

Note

ISSUE: The Motif drag and drop protocol creates and maps such a window (at -100, -100, size 10x10) to "cache frequently needed data on window properties to reduce roundtrip server requests." Proposed solution: we could only send an error if the window is visible, which would require checking in, MapWindow, ConfigureWindow, and ReparentWindow.

Theft: resource ID guessing (parent, background-pixmap, border-pixmap, colormap, and cursor).

Defense: send Window, Pixmap, Colormap, or Cursor error.

Denial of service: create windows until the server runs out of memory.

Defense: quotas.

Also Chapter 3, *Security concerns with specific window attributes*

ChangeWindowAttributes

Alteration: change the attributes of another client's window.

Theft: select for events on another client's window.

Defense for both of the above: send Window error.

Note

ISSUE: The Motif drop protocol states that "the initiator should select for DestroyNotify on the destination window such that it is aware of a potential receiver crash." This will be a problem if the initiator is an untrusted window and the destination is trusted. Can the server, perhaps with the help of the security manager, recognize that a drop is in progress and allow the DestroyNotify event selection in this limited case?

Note

ISSUE: The Motif pre-register drag protocol probably requires the initiator to select for Enter/LeaveNotify on all top-level windows. Same problem as the previous issue.

Theft: resource ID guessing (background-pixmap, border-pixmap, colormap, and cursor).

Defense: send Pixmap, Colormap, or Cursor error.

Also Chapter 3, *Security concerns with specific window attributes*

GetWindowAttributes

Theft: get the attributes of another client's window.

Theft: resource ID guessing (window).

Defense for both of the above: send Window error.

DestroyWindow, DestroySubwindows

Destruction: destroy another client's window.

Theft: resource ID guessing (window).

Defense for both of the above: send Window error.

ChangeSaveSet

Alteration: cause another client's windows to be reparented to the root when this client disconnects (only if the other client's windows are subwindows of this client's windows).

Defense: process the request normally. The trusted client gives away some of its protection by creating a subwindow of an untrusted window.

Theft: resource ID guessing (window).

Defense: send Window error.

MapWindow

Spoofing: map a window that is designed to resemble a window of another client. Additional requests will probably be needed to complete the illusion.

Defense:

We consider spoofing to be a significant danger only if the user is convinced to interact with the spoof window. The defense centers on providing enough information to enable the user to know where keyboard, mouse, and extension device input is going. To accomplish this, the server will cooperate with the security manager, an external process. The server will provide the following facilities to the security manager:

1. A way to create a single window that is unobscurable by any window of any other client, trusted or untrusted. It needs to be unobscurable so that it is spoof-proof.

> ### Note
>
> ISSUE: is a weaker form of unobscurability better? Should the window be obscurable by trusted windows, for example?

> ### Note
>
> ISSUE: does unobscurable mean that it is a child of the root that is always on top in the stacking order?

2. A way to determine if a given window ID belongs to an untrusted client.

The security manager will need to select for the existing events FocusIn, FocusOut, EnterNotify, LeaveNotify, DeviceFocusIn, and DeviceFocusOut on all windows to track what window(s) the user's input is going to. Using the above server facilities, it can reliably display the trusted/untrusted status of all clients currently receiving input.

> ### Note
>
> ISSUE: is it too much to ask the security manager to select for all these events on every window? Do we need to provide new events that you select for *on the device* that tell where the device is focused?

None of this should have any application impact.

The unobscurable window may be tricky to implement. There is already some machinery in the server to make an unobscurable window for the screen saver, which may help but may also get in the way now that we have to deal with two unobscurable windows.

Window Operations

Specifically, ReparentWindow, MapWindow, MapSubwindows, UnmapWindow, UnmapSubwindows, ConfigureWindow, and CirculateWindow.

Alteration: manipulate another client's window.

Theft: resource ID guessing (window, sibling).

Defense for both of the above: send a Window error unless it is a root window, in which case we should send an Access error.

GetGeometry

Theft: get the geometry of another client's drawable.

Theft: resource ID guessing (drawable).

Defense for both of the above: send Drawable error. However, root windows will be allowed.

QueryTree

Theft: resource ID guessing (window).

Defense: send Window error.

Theft: discover window IDs that belong to other clients.

Defense: For the child windows, censor the reply by removing window IDs that belong to trusted clients. Allow the root window to be returned. For the parent window, if it belongs to a trusted client, return the closest ancestor window that belongs to an untrusted client, or if such a window does not exist, return the root window for the parent window.

Note

ISSUE: will some applications be confused if we filter out the window manager frame window(s), or other windows between the queried window and the root window?

Note

ISSUE: the Motif drag protocol (both preregister and dynamic) needs to be able to locate other top-level windows for potential drop sites. See also the section called “Access to Server Resources” .

InternAtom

Theft: discover atom values of atoms interned by other clients. This lets you determine if a specific set of atoms has been interned, which may lead to other inferences.

Defense: This is a minor form of theft. Blocking it will interfere with many types of inter-client communication. We propose to do nothing about this threat.

Denial of service: intern atoms until the server runs out of memory.

Defense: quotas.

GetAtomName

Theft: discover atom names of atoms interned by other clients. This lets you determine if a specific set of atoms has been interned, which may lead to other inferences.

Defense: This is a minor form of theft. We propose to do nothing about this threat.

ChangeProperty

Alteration: change a property on another client's window or one that was stored by another client.

Theft: resource ID guessing (window).

Defense for both of the above: send Window error.

Note

ISSUE: Selection transfer requires the selection owner to change a property on the requestor's window. Does the security manager get us out of this? Does the server notice the property name and window passed in ConvertSelection and temporarily allow that window property to be written?

Note

ISSUE: should certain root window properties be writable?

Denial of service: store additional property data until the server runs out of memory.

Defense: quotas.

DeleteProperty

Destruction: delete a property stored by another client.

Theft: resource ID guessing (window).

Defense for both of the above: send Window error.

GetProperty

Theft: get a property stored by another client.

Theft: resource ID guessing (window).

Defense for both of the above: send Window error.

Note

ISSUE: should certain root window properties be readable? Proposed answer: yes, some configurable list. Do those properties need to be polyinstantiated?

Note

ISSUE: Motif drag and drop needs to be able to read the following properties: WM_STATE to identify top-level windows, _MOTIF_DRAG_WINDOW on the root window, _MOTIF_DRAG_TARGETS on the window given in the _MOTIF_DRAG_WINDOW property, and _MOTIF_DRAG_RECEIVER_INFO on windows with drop sites. Additionally, some properties are needed that do not have fixed names.

RotateProperties

Alteration: rotate properties stored by another client.

Theft: resource ID guessing (window).

Defense for both of the above: send Window error.

ListProperties

Theft: list properties stored by another client.

Theft: resource ID guessing (window).

Defense for both of the above: send Window error.

Note

ISSUE: should certain root window properties be listable?

SetSelectionOwner

Theft: Steal ownership of a selection.

Denial of service: do this repeatedly so that no other client can own the selection.

Defense for both of the above: have a configurable list of selections that untrusted clients can own. For other selections, treat this request as a no-op.

Note

ISSUE: how does the security manager get involved here? Is it the one that has the configurable list of selections instead of the server?

Theft: resource ID guessing (window).

Defense: send Window error.

GetSelectionOwner

Theft: discover the ID of another client's window via the owner field of the reply.

Defense: if the selection is on the configurable list mentioned above, return the root window ID, else return None.

Note

ISSUE: how does the security manager get involved here?

ConvertSelection

Theft: this initiates a selection transfer (see the ICCCM) which sends the selection contents from the selection owner, which may be another client, to the requesting client.

Defense: since in many cases ConvertSelection is done in direct response to user interaction, it is probably best not to force it to fail, either silently or with an error. The server should rely on the security manager to assist in handling the selection transfer.

Theft: resource ID guessing (requestor).

Defense: send Window error.

SendEvent

A client can use SendEvent to cause events of any type to be sent to windows of other clients. Similarly, a client could SendEvent to one of its own windows with propogate set to True and arrange for the event to be propogated up to a window it does not own. Clients can detect events generated by SendEvent, but we cannot assume that they will.

Defense: ignore this request unless the event being sent is a ClientMessage event, which should be sent normally so that selection transfer, Motif drag and drop, and certain input methods have a chance at working.

Note

ISSUE: does allowing all ClientMessages open up too big a hole?

Theft: resource ID guessing (window).

Defense: send Window error.

Keyboard and Pointer Grabs

Specifically, GrabKeyboard, GrabPointer, GrabKey, and GrabButton.

Denial of service/Theft: take over the keyboard and pointer. This could be viewed as denial of service since it prevents other clients from getting keyboard or mouse input, or it could be viewed as theft since the user input may not have been intended for the grabbing client.

Defense: provide a way to break grabs via some keystroke combination, and have a status area that shows which client is getting input. (See the section called “MapWindow”).

Theft: resource ID guessing (grab-window, confine-to, cursor).

Defense: send Window or Cursor error.

ChangeActivePointerGrab

Theft: resource ID guessing (cursor).

Defense: send Cursor error.

GrabServer

Denial of service: a client can grab the server and not let go, locking out all other clients.

Defense: provide a way to break grabs via some keystroke combination.

QueryPointer

Theft: A client can steal pointer motion and position, button input, modifier key state, and possibly a window of another client with this request.

Defense: if the querying client doesn't have the pointer grabbed, and the pointer is not in one of its windows, the information can be zeroed.

Theft: resource ID guessing (window).

Defense: send Window error.

GetMotionEvents

Theft: steal pointer motion input that went to other clients.

Defense: ideally, the server would return only pointer input that was not delivered to any trusted client. The implementation effort to do that probably outweighs the marginal benefits. Instead, we will always return an empty list of motion events to untrusted clients.

Theft: resource ID guessing (window).

Defense: send Window error.

TranslateCoordinates

Theft: discover information about other clients' windows: position, screen, and possibly the ID of one of their subwindows.

Defense: send an error if src-window or dst-window do not belong to the requesting client.

Theft: resource ID guessing (src-window, dst-window).

Defense: send Window error.

WarpPointer

A client can cause pointer motion to occur in another client's window.

Denial of service: repeated pointer warping prevents the user from using the mouse normally.

Defense for both of the above: if the querying client doesn't have the pointer grabbed, and the pointer is not in one of its windows, treat the request as a no-op.

Theft: resource ID guessing (src-window, dst-window).

Defense: send Window error.

SetInputFocus

Theft: a client can use this request to make one of its own windows have the input focus (keyboard focus). The user may be unaware that keystrokes are now going to a different window.

Denial of service: repeatedly setting input focus prevents normal use of the keyboard.

Defense for both of the above: only allow untrusted clients to SetInputFocus if input focus is currently held by another untrusted client.

Note

ISSUE: this will break clients using the Globally Active Input model described in section 4.1.7 of the ICCCM.

Theft: resource ID guessing (focus).

Defense: send Window error.

GetInputFocus

Theft: the reply may contain the ID of another client's window.

Defense: return a focus window of None if a trusted client currently has the input focus.

QueryKeymap

Theft: poll the keyboard with this to see which keys are being pressed.

Defense: zero the returned bit vector if a trusted client currently has the input focus.

Font Requests

Specifically, OpenFont, QueryFont, ListFonts, ListFontsWithInfo, and QueryTextExtents.

Theft: discover font name, glyph, and metric information about fonts that were provided by another client (by setting the font path). Whether it is theft to retrieve information about fonts from the server's initial font path depends on whether or not you believe those fonts, by their existence in the initial font path, are intended to be globally accessible by all clients.

Defense:

Maintain two separate font paths, one for trusted clients and one for untrusted clients. They are both initialized to the default font path at server reset. Subsequently, changes to one do not affect the other. Since untrusted clients will not see font path elements added by trusted clients, they will not be able to access any fonts provided by those font path elements.

Theft: resource ID guessing (font) (QueryFont and QueryTextExtents only).

Defense: send Font error.

Denial of service: open fonts until the server runs out of memory (OpenFont only).

Defense: quotas.

CloseFont

Destruction: close another client's font.

Defense: send Font error.

SetFontPath

Denial of service: change the font path so that other clients cannot find their fonts.

Alteration: change the font path so that other clients get different fonts than they expected.

Defense for both of the above: separate font paths for trusted and untrusted clients, as described in the Font Requests section.

Note

ISSUE: the printing project considered per-client font paths and concluded that it was very difficult to do. We should look at this aspect of the print server design to see if we can reuse the same scheme. We should also try to reconstruct what was so difficult about this; it doesn't seem that hard on the surface.

GetFontPath

Theft: retrieve font path elements that were set by other clients.

Use knowledge from font path elements to mount other attacks, e.g., attack a font server found in the font path.

Defense for both of the above: separate font paths for trusted and untrusted clients, as described in the Font Requests section.

CreatePixmap

Theft: resource ID guessing (drawable).

Defense: send Drawable error.

Denial of service: create pixmaps until the server runs out of memory.

Defense: quotas.

FreePixmap

Destruction: destroy another client's pixmap.

Defense: send Pixmap error.

CreateGC

Theft: resource ID guessing (drawable, tile, stipple, font, clip-mask).

Defense: send Drawable, Pixmap, or Font error.

Denial of service: create GCs until the server runs out of memory.

Defense: quotas.

CopyGC

Theft: copy GC values of another client's GC.

Alteration: copy GC values to another client's GC.

Defense for both of the above: send GC error.

ChangeGC, SetDashes, SetClipRectangles

Alteration: change values of another client's GC.

Theft: resource ID guessing (gc, tile, stipple, font, clip-mask) (last four for ChangeGC only).

Defense for both of the above: send GC error.

FreeGC

Destruction: destroy another client's GC.

Defense: send GC error.

Drawing Requests

Specifically, ClearArea, CopyArea, CopyPlane, PolyPoint, PolyLine, PolySegment, PolyRectangle, PolyArc, FillPoly, PolyFillRectangle, PolyFillArc, PutImage, PolyText8, PolyText16, ImageText8, and ImageText16.

Alteration: draw to another client's drawable.

Theft: resource ID guessing: ClearArea - window; CopyArea, CopyPlane - src-drawable, dst-drawable, gc; all others - drawable, gc.

Defense for both of the above: send appropriate error.

Note

ISSUE: The Motif preregister drag protocol requires clients to draw into windows of other clients for drag-over/under effects.

Spoofing: draw to a window to make it resemble a window of another client.

Defense: see the section called “MapWindow” .

GetImage

Theft: get the image of another client's drawable.

Theft: resource ID guessing (drawable).

Defense: send Drawable error.

Theft: get the image of your own window, which may contain pieces of other overlapping windows.

Defense: censor returned images by blotting out areas that contain data from trusted windows.

CreateColormap

Theft: resource ID guessing (window).

Defense: send Colormap error.

Denial of service: create colormaps with this request until the server runs out of memory.

Defense: quotas.

FreeColormap

Destruction: destroy another client's colormap.

Defense: send Colormap error.

CopyColormapAndFree

Theft: resource ID guessing (src-map).

Defense: send Colormap error. However, default colormaps will be allowed.

Note

ISSUE: must untrusted applications be allowed to use standard colormaps? (Same issue for ListInstalledColormaps, Color Allocation Requests, FreeColors, StoreColors, StoreNamedColor, QueryColors, and LookupColor.)

Denial of service: create colormaps with this request until the server runs out of memory.

Defense: quotas.

InstallColormap, UninstallColormap

Theft: resource ID guessing.

Defense: send Colormap error.

Denial of service: (un)install any colormap, potentially preventing windows from displaying correct colors.

Defense: treat this request as a no-op. Section 4.1.8 of the ICCCM states that (un)installing colormaps is the responsibility of the window manager alone.

Note

ISSUE: the ICCCM also allows clients to do colormap installs if the client has the pointer grabbed. Do we need to allow that too?

ListInstalledColormaps

Theft: resource ID guessing (window).

Defense: send Colormap error.

Theft: discover the resource ID of another client's colormap from the reply.

Defense: remove the returned colormap IDs; only let through default colormaps and colormaps of untrusted clients.

Color Allocation Requests

Specifically, AllocColor, AllocNamedColor, AllocColorCells, and AllocColorPlanes.

Alteration/Denial of service: allocate colors in another client's colormap. It is denial of service if the owning client's color allocations fail because there are no cells available. Otherwise it is just alteration.

Theft: resource ID guessing (cmap).

Defense for both of the above: send Colormap error. However, default colormaps will be allowed.

FreeColors

Theft: resource ID guessing (cmap).

Defense: send Colormap error. However, default colormaps will be allowed.

StoreColors, StoreNamedColor

Alteration: change the colors in another client's colormap.

Theft: resource ID guessing (cmap).

Defense for both of the above: send Colormap error. However, default colormaps will be allowed.

QueryColors, LookupColor

Theft: retrieve information about the colors in another client's colormap.

Theft: resource ID guessing (cmap).

Defense for both of the above: send Colormap error. However, default colormaps will be allowed.

CreateCursor, CreateGlyphCursor

Theft: resource ID guessing (source, mask or source-font, mask-font).

Defense: send Pixmap or Font error. However, the default font will be allowed.

Denial of service: create cursors until the server runs out of memory.

Defense: quotas.

FreeCursor

Destruction: free another client's cursor.

Defense: send Cursor error.

RecolorCursor

Alteration: recolor another client's cursor.

Theft: resource ID guessing (cursor).

Defense for both of the above: send Cursor error.

QueryBestSize

Theft: resource ID guessing (drawable).

Defense: send Drawable error.

ListExtensions, QueryExtension

Determine the extensions supported by the server, and use the list to choose extension-specific attacks to attempt.

Defense: extensions will have a way to tell the server whether it is safe for untrusted clients to use them. These requests will only return information about extensions that claim to be safe.

Keyboard configuration requests

Specifically, ChangeKeyboardControl, ChangeKeyboardMapping, and SetModifierMapping.

Alteration: change the keyboard parameters that were established by another client.

Denial of service: with ChangeKeyboardControl, disable auto-repeat, key click, or the bell. With ChangeKeyboardMapping or SetModifierMapping, change the key mappings so that the keyboard is difficult or impossible to use.

Defense for both of the above: treat these requests as a no-op.

Keyboard query requests

Specifically, GetKeyboardControl, GetKeyboardMapping, and GetModifierMapping.

Theft: get keyboard information that was established by another client.

Defense: This is a minor form of theft. We propose to do nothing about this threat.

ChangePointerControl, SetPointerMapping

Alteration: change the pointer parameters that were established by another client.

Denial of service: set the pointer parameters so that the pointer is difficult or impossible to use.

Defense for both of the above: treat these requests as a no-op.

GetPointerControl, GetPointerMapping

Theft: get pointer parameters that were established by another client.

Defense: This is a minor form of theft. We propose to do nothing about this threat.

SetScreenSaver

Alteration: change the screen saver parameters that were established by another client.

Denial of service: set the screen saver parameters so that the screen saver is always on or always off.

Defense for both of the above: treat these requests as a no-op.

GetScreenSaver

Theft: get screen saver parameters that were established by another client.

Defense: This is a minor form of theft. We propose to do nothing about this threat.

ForceScreenSaver

Denial of service: repeatedly activate the screen saver so that the user cannot see the screen as it would look when the screen saver is off.

Denial of service: repeatedly reset the screen saver, preventing it from activating.

Defense for both of the above: treat these requests as a no-op.

ChangeHost

Most servers already have some restrictions on which clients can use this request, so whether the following list applies is implementation dependent.

Denial of service: remove a host from the list, preventing clients from connecting from that host.

Add a host to the list. Clients from that host may then launch other attacks of any type.

Defense for both of the above: return Access error.

ListHosts

Theft: steal host identities and possibly even user identities that are allowed to connect.

Launch attacks of any type against the stolen host/user identities.

Defense for both of the above: return only untrusted hosts.

SetAccessControl

Most servers already have some restrictions on which clients can use this request, so whether the following list applies is implementation dependent.

Alteration: change the access control value established by some other client.

Disable access control, allowing clients to connect who would normally not be able to connect. Those clients may then launch other attacks of any type.

Defense for both of the above: return Access error.

SetCloseDownMode

Denial of service: set the close-down mode to RetainPermanent or RetainTemporary, then disconnect. The server cannot reuse the resource-id-base of the disconnected client, or the memory used by the retained resources, unless another client issues an appropriate KillClient to cancel the retainment. The server has a limited number of resource-id-bases, and when they are exhausted, it will be unable to accept new client connections.

Defense: treat this request as a no-op.

KillClient

Destruction/Denial of service: kill another currently connected client.

Destruction: kill a client that has terminated with close-down mode of RetainTemporary or RetainPermanent, destroying all its retained resources.

Destruction: specify AllTemporary as the resource, destroying all resources of clients that have terminated with close-down mode RetainTemporary.

Defense for all of the above: return Value error.

Clean Requests

Other than denial of service caused by flooding, these requests have no known security concerns: AllowEvents, UngrabPointer, UngrabButton, UngrabKeyboard, UngrabKey, UngrabServer, NoOperation, and Bell.

Chapter 5. Events

The only threat posed by events is theft. Selecting for events on another client's resources is always theft. We restrict further analysis by assuming that the client only selects for events on its own resources, then asking whether the events provide information about other clients.

KeymapNotify

Theft: the state of the keyboard can be seen when the client does not have the input focus. This is possible because a KeymapNotify is sent to a window after every EnterNotify even if the window does not have input focus.

Defense: zero the returned bit vector if a trusted client currently has the input focus.

Expose

Theft: discover where other clients' windows overlap your own. For instance, map a full-screen window, lower it, then raise it. The resulting exposes tell you where other windows are.

Defense: about the only thing you could do here is force backing store to be used on untrusted windows, but that would probably use too much server memory. We propose to do nothing about this threat.

GraphicsExposure

Theft: discover where other clients' windows overlap your own. For instance, use CopyArea to copy the entire window's area exactly on top of itself. The resulting GraphicsExposures tell you where the window was obscured.

Defense: see Expose above. We propose to do nothing about this threat.

VisibilityNotify

Theft: this event provides crude positional information about other clients, though the receiver cannot tell which other clients.

Defense: The information content of this event is very low. We propose to do nothing about this threat.

ReparentNotify

Theft: the parent window may belong to some other client (probably the window manager).

Defense: If the parent window belongs to a trusted client, return the closest ancestor window that belongs to an untrusted client, or if such a window does not exist, return the root window for the parent window.

Note

ISSUE: what is the application impact?

ConfigureNotify

Theft: the above-sibling window may belong to some other client.

Defense: return None for the above-sibling window if it belongs to a trusted client.

Note

ISSUE: what is the application impact?

ConfigureRequest

Theft: the sibling window may belong to some other client.

Defense: return None for the sibling window if it belongs to a trusted client.

ISSUE: what is the application impact?

SelectionClear

Theft: the owner window may belong to some other client.

Defense: return None for the owner window if it belongs to a trusted client.

SelectionRequest

Theft: the requestor window may belong to some other client.

Defense: Blocking this event or censoring the window would prevent selection transfers from untrusted clients to trusted clients from working. We propose to do nothing in the server about this threat. The security manager may reduce the exposure of trusted window IDs by becoming the owner of all selections.

MappingNotify

Theft: discover keyboard, pointer, or modifier mapping information set by another client.

Defense: Any tampering with this event will cause clients to have an inconsistent view of the keyboard or pointer button configuration, which is likely to confuse the user. We propose to do nothing about this threat.

Chapter 6. Errors

There appear to be no threats related to procotol errors.

Chapter 7. Future Work

The next steps are resolve the items marked ISSUE and to decide if the defenses proposed are reasonable. Discussion on the security@x.org mailing list, prototyping, and/or starting the implementation should help answer these questions.

Chapter 8. References

Bellcore, "Framework Generic Requirements for X Window System Security," Technical Advisory FA-STS-001324, Issue 1, August 1992.

Dardailler, Daniel, "Motif Drag And Drop Protocol," unpublished design notes.

Kahn, Brian L., "Safe Use of X WINDOW SYSTEM protocol Across a Firewall", unpublished draft, The MITRE Corporation, 1995.

Rosenthal, David S. H., "LINX - a Less INsecure X server," Sun Microsystems, 29th April 1989.

Rosenthal, David and Marks, Stuart W., "Inter-Client Communication Conventions Manual Version 2.0," ftp://ftp.x.org/pub/R6.1/xc/doc/hardcopy/ICCCM/icccm.PS.Z

Scheifler, Robert W., "X Window System Protocol," ftp://ftp.x.org/pub/R6.1/xc/doc/hardcopy/XProtocol/proto.PS.Z

Treese, G. Winfield and Wolman, Alec, "X Through the Firewall, and Other Application Relays," Digital Equipment Corporation Cambridge Research Lab, Technical Report Series, CRL 93/10, May 3, 1993.