Security Extension Server Design Draft

X Consortium Standard

David Wiggins

Security Extension Server Design Draft: X Consortium Standard

by David Wiggins

June 27, 2010

Version 3.0

Abstract

This paper describes the implementation strategy used to implement various pieces of the SECURITY Extension.

Table of Contents

1. Generate Authorization Request 1
2. Client Connection 2
3. Client disconnection 3
4. Resource ID Security 4
5. Extension Security 5

Chapter 1. Generate Authorization Request

The major steps taken to execute this request are as follows.

Sanity check arguments. The interesting one is the group, which must be checked by some other module(s), initially just the embedding extension. Use a new callback for this. The callback functions will be passed a small structure containing the group ID and a Boolean value which is initially false. If any of the callbacks recognize the ID, they should set the boolean to true. If after the callbacks have been called the boolean is false, return an error, since nobody recognized it.

Use the existing Xkey library function XkeyGenerateAuthorization to generate the new authorization.

Use the existing os layer function AddAuthorization to add the new authorization to the server's internal database.

Use the existing os layer function AuthorizationToID to retrieve the authorization ID that the os layer assigned to the new authorization.

Change the os layer to use authorization IDs allocated from the server's ID range via FakeClientID(0) instead of using a simple incrementing integer. This lets us use the resource database to attach additional information to an authorization without needing any changes to os data structures.

Add the authorization ID as a server resource. The structure for an authorization resource will contain the timeout, trust-level, and group sent in the request, a reference count of how many clients are connected with this authorization, a timer pointer, and time-remaining counter.

Return the authorization ID and generated auth data to the client.

Chapter 2. Client Connection

The Security extension needs to be aware of new client connections primarily so that it copy the trust-level of the authorization that was used to the client structure. The trust-level is needed in the client structure because it will be accessed frequently to make access control decisions for the client. We will use the existing ClientStateCallback to catch new client connections.

We also need to copy the authorization ID into the client structure. The authorization ID is already stored in an os private hung from the client, and we will add a new os function AuthorizationIDOfClient to retrieve it. However, when a client disconnects, this os private is already gone before ClientStateCallbacks are called. We need the authorization ID at client disconnect time for reasons described below.

Now that we know what needs to be done and why, let's walk through the sequnce of events.

When a new client connects, get the authorization ID with AuthorizationIDOfClient, store it in the client, then pass that ID to LookupIDByType to find the authorization. If we get a non-NULL pointer back, this is a generated authorization, not one of the predefined ones in the server's authority file. In this case, increment the authorization's reference count. If the reference count is now 1, cancel the timer for this authorization using the trivial new os layer function TimerCancel. Lastly, copy the trust-level of this authorization into the client structure so that it can be reached quickly for future access control decisions.

The embedding extension can determine the group to use for a new client in the same way that we determined the trust level: get the authorization ID, look it up, and if that succeeds, pluck the group out of the returned authorization structure.

Chapter 3. Client disconnection

Use the existing ClientStateCallback to catch client disconnections. If the client was using a generated authorization, decrement its reference count. If the reference count is now zero, use the existing os layer function TimerSet to start a timer to count down the timeout period for this authorization. Record the timer ID for this authorization. When the timer fires, the authorization should be freed, removing all traces of it from the server.

There is a slight complication regarding the timeout because the timer interface in the server allows for 32 bits worth of milliseconds, while the timeout specified in GenerateAuthorization has 32 bits worth of seconds. To handle this, if the specified time is more than the timer interface can handle, the maximum possible timeout will be set, and time-remaining counter for this authorization will be used to track the leftover part. When the timer fires, it should first check to see if there is any leftover time to wait. If there is, it should set another timer to the minimum of (the maximum possible timeout) and the time remaining, and not do the revocation yet.

Chapter 4. Resource ID Security

To implement the restriction that untrusted clients cannot access resources of trusted clients, we add two new functions to dix: SecurityLookupIDByType and SecurityLookupIDByClass. Hereafter we will use SecurityLookupID to refer to both functions. In addition to the parameters of the existing LookupID functions, these functions also take a pointer to the client doing the lookup, and an access mode that conveys a high-level idea of what the client intends to do with the resource (currently just read, write, destroy, and unknown). Passing NullClient for the client turns off access checks. SecurityLookupID can return NULL for two reasons: the resource doesn't exist, or it does but the client isn't allowed to access it. The caller cannot tell the difference. Most places in dix call these new lookup functions instead of the old LookupID, which continue to do no access checking. Extension "Proc" functions should probably use SecurityLookupID, not LookupID. Ddxen can continue to use LookupID.

Inside SecurityLookupID, the function client -> CheckAccess is called passing the client, resource id, resource type/class, resource value, and access mode. CheckAccess returns the resource value if access is allowed, else it returns NULL. The entire resource ID security policy of the Security extension can be replaced by plugging in your own access decision function here. This in combination with the access mode parameter should be enough to implement a more traditional DAC (discretionary access control) policy.

Since we need client and access mode information to do access controlled resource lookups, we add (and use) several other macros and functions that parallel existing ones with the addition of the missing information. The list includes SECURITY_VERIFY_GC, SECURITY_VERIFY_DRAWABLE, SECURITY_VERIFY_GEOMETRABLE, SecurityLookupWindow, SecurityLookupDrawable, and dixChangeGC. The dixChangeGC interface is worth mentioning because in addition to a client parameter, we introduce a pointer-to-union parameter that should let us eliminate the warnings that some compilers give when you assign small integers to pointers, as the DoChangeGC interface required. For more details, see the comment preceding dixChangeGC in <dix/gc.c >.

If XCSECURITY is not defined (the Security extension is not being built), the server uses essentially the same code as before for resource lookups.

Chapter 5. Extension Security

A new field in the ExtensionEntry structure, Bool secure, tells whether the extension is considered secure. It is initialized to FALSE by AddExtension. The following new dix function can be used to set the secure field:

`void` **`DeclareExtensionSecurity`**`(` *`*extname`*`,` *`secure`*`);`

The name of the extension and the desired value of the secure field are passed. If an extension is secure, a call to this function with secure = TRUE will typically appear right after the call to `AddExtension`. `DeclareExtensionSecurity` should be called during server reset. It should not be called after the first client has connected. Passing the name of an extension that has not been initialized has no effect (the secure value will not be remembered in case the extension is later initialized).

For untrusted clients, `ProcListExtensions` omits extensions that have secure = FALSE, and `ProcQueryExtension` reports that such extensions don't exist.

To prevent untrusted clients from using extensions by guessing their major opcode, one of two new Proc vectors are used by untusted clients, `UntrusedProcVector` and `SwappedUntrustedProcVector`. These have the same contents as `ProcVector` and `SwappedProcVector` respectively for the first 128 entries. Entries 128 through 255 are initialized to ProcBadRequest. If `DeclareExtensionSecurity` is called with secure = TRUE, that extension's dispatch function is plugged into the appropriate entry so that the extension can be used. If `DeclareExtensionSecurity` is called with secure = FALSE, the appropriate entry is reset to ProcBadRequest.

Now we can explain why `DeclareExtensionSecurity` should not be called after the first client connects. In some cases, the Record extension gives clients a private copy of the proc vector, which it then changes to intercept certain requests. Changing entries in `UntrusedProcVector` and `SwappedUntrustedProcVector` will have no effect on these copied proc vectors. If we get to the point of needing an extension request to control which extensions are secure, we'll need to invent a way to get those copied proc vectors changed.