Authorization Plug-in Reference

Security > Authorization

2007-05-15

Contents

Listings

Authorization Plug-in Reference

Framework:	Security/Security.h
Declared in	AuthorizationPlugin.h

Overview

The Authorization Plug-in API enables you to create plug-ins that can participate in authorization decisions.

Authorization Plug-ins are available starting in Mac OS X v10.4.

You should read this document if you need to extend Mac OS X authorization services to perform authorizations in a new way or to implement a new policy that is too complex to be implemented entirely with the authorization policy database.

Organization of This Document

This document consists of an introduction and a reference to the authorization plug-in API. It contains the following sections:

- "About Authorization Plug-ins" (page 8) gives a brief introduction to the purpose and use of authorization plug-ins and gives basic instructions for installing one.
- "Authorization Plug-in Functions" (page 10) describes the functions that your plug-in can call to communicate with the authorization engine. You use these functions to set and get data and to report the status of your plug-in.
- "Functions Implemented By the Plug-in" (page 18) describes the functions that must be implemented by your plug-in. These functions are called by the authorization engine in order to install the plug-in, implement and execute the plug-in's authorization mechanisms, and deactivate or remove the plug-in.
- "Authorization Plug-in Data Types" (page 23) describes the structures and other data types defined in `AuthorizationPlugin.h`.
- "Authorization Plug-in Constants" (page 27) describes the constants defined in `AuthorizationPlugin.h`.
- "Authorization Plug-in Result Codes" (page 29) lists the result codes used by authorization plug-ins.

See Also

For more information about Mac OS X authorization services, see the following documents:

- Authorization Services Programming Guide, which describes authorization services and several components of authorization services, such as the authorization policy data base and the Security Server. This document also provides some sample code.
- Authorization Services Reference, which describes the Authorization Services API.

About Authorization Plug-ins

To install an authorization plug-in, you write the plug-in using the API described in this document, install the bundle in `/System/Library/CoreServices/SecurityAgentPlugins`, and use the `AuthorizationRightSet` function to add an entry to the authorization policy database that references the plug-in. The authorization policy database contains a set of rules that the Security Server uses to authorize rights for a user. Most of the rules directly specify the criteria to allow or deny access; however, some reference external code (referred to as *authorization mechanisms*) that define the behavior. The authorization database is described in The Policy Database section of *Authorization Services Programming Guide*. To invoke a plug-in, you pass the name of the database entry that references that plug-in in the `rights` parameter of the `AuthorizationCopyRights` function.

A typical use for authorization plug-ins is to implement policies that are not included in the standard authorization configuration. For example, you could write a plug-in that authorizes a user to send a fax by requiring a personal identification number (PIN) for a specific fax machine.

Important: If your plug-in displays a window before the user has logged in, you must set the `KHIWindowBitCanBeVisibleWithoutLogin` flag on the window. See `AuthorizationPluginCreate` (page 18) for more information on setting this flag.

A plug-in's main entry point must be the function `AuthorizationPluginCreate` (page 18), which exchanges the plug-in's interface (`AuthorizationPluginInterface` (page 27)) and the authorization interface of the Security Server (`AuthorizationCallbacks` (page 26)).

When you add a policy to the authorization policy database, it can refer to any number of plug-ins. Each plug-in includes one or more authorization mechanisms, where a mechanism is a code module that performs one step in the authorization process.

For example, if you wrote a policy for sending faxes that required users to select the fax machine they wanted to use and enter a PIN for that machine, you might name the policy `com.ifoo.ifax.send`. To implement the policy, you could write a plug-in called `SendFaxPlugin` that contains two mechanisms: `SelectFaxMachine` and `GetUserPIN`. You would add your plug-in code to the folder `/System/Library/CoreServices/SecurityAgentPlugins` as a bundle called `SendFaxPlugin.bundle` and you would use the `AuthorizationRightSet` function to add the lines shown in Listing 1 to the authorization policy database:

Listing 1 Plug-in entry in policy database

```
<key>com.ifoo.ifax.send</key>
        <dict>
            <key>class</key>
            <string>evaluate-mechanisms</string>
            <key>comment</key>
            <string>Rule to evaluate whether user has right  to
                    use a specific fax machine.
            </string>
            <key>mechanisms</key>
            <array>
                <string>SendFaxPlugin:SelectFaxMachine</string>
                <string>SendFaxPlugin:GetUserPIN</string>
            </array>
        </dict>
```

Notice that each plug-in is identified by the name of the plug-in, a colon, and the name of the mechanism; for example `SendFaxPlugin:SelectFaxMachine` where `SelectFaxMachine` is a mechanism in the plug-in `SendFaxPlugin`.The keys used in the dictionary entry are listed in the files `AuthorizationTags.h` and `AuthorizationTagsPriv.h`. (Note that `AuthorizationTagsPriv.h` is not part of the public API. Apple reserves the right to change this file or its contents with future releases.)

The Security Server loads plug-ins into a separate process—a plug-in host—to isolate the process of authorization from the client. There are two plug-in hosts:

- One runs as an anonymous user and can be used to communicate with the user, for example to ask for a password.
- One runs with root privileges to perform privileged operations.

In this document, the portion of the Security Server that deals with authorization and authentication, together with the plug-in hosts, is referred to as the *authorization engine*.

To have a specific mechanism run with root privileges, add a comma and the word `privileged` to the mechanism name; for example:

```
<string>SendFaxPlugin:ChangeUserPIN,privileged</string>
```

Important: Authorization plug-ins that put up a GUI or otherwise connect to the window server cannot run as privileged. Note that running GUI code as root is a bad idea in general, because GUI code links in many libraries, any of which could contain security vulnerabilities.

When the authorization engine needs an authorization decision based on a policy that belongs to the plug-in, the authorization engine calls each mechanism belonging to that policy in turn, in the order they are listed in the policy database. For each mechanism, the authorization engine calls the plug-in's `MechanismInvoke` (page 22) function, passing the *plug-in name*:*mechanism name* for that mechanism.

The mechanism calls the `SetResult` (page 17) function to report the authorization decision. The authorization engine does not consider the authorization complete and approved until all the mechanisms have returned a positive (`kAuthorizationResultAllow`) authorization decision, one of the mechanisms has returned a negative (`kAuthorizationResultDeny`) decision, the maximum number of retries has been reached (`kAuthorizationResultUndefined`), or the user has canceled the attempt (`kAuthorizationResultUserCanceled`).

Mechanisms in the authorization can communicate auxiliary information by setting and getting hints and context data. Hints are data values for use during authorization; for example, you can use a hint to pass an intermediate value from one mechanism to a subsequent mechanism. They are not preserved as part of the authorization result. Context data is information that can be useful to an application, such as a user name entered by the user during the authorization process. Context data can be added, read, or modified by each mechanism in the authorization and is preserved by the Security Server. Context data can also be made available to the authorization client after authorization is complete. See `SetHintValue` (page 16) and `SetContextValue` (page 15) for more information on hints and context data.

When the authorization plug-in sets context data, it tags the data with a flag that specifies whether the information should be returned to the authorization client upon request (by using the `AuthorizationCopyInfo` function) or whether it's restricted to the mechanisms involved in the authorization.

Functions

Your authorization plug-in communicates with the authorization engine through the engine's callback functions. These functions are declared in the `AuthorizationCallbacks` (page 26) structure passed to your plug-in through the `AuthorizationPluginCreate` (page 18) function.

DidDeactivate

Report the successful deactivation of an authorization mechanism.

```
OSStatus (*DidDeactivate)(
    AuthorizationEngineRef inEngine,
);
```

Parameters

`inEngine`

An opaque handle that is passed to your plug-in when the authorization engine calls your `MechanismCreate` (page 20) function.

Return Value

A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion

You must call this function after deactivating your authorization mechanism in response to a call to your `MechanismDeactivate` (page 21) function. The authorization engine waits for confirmation that all mechanisms have deactivated before continuing.

The authorization engine sends you the entry point to the `DidDeactivate` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Availability

Available in Mac OS X v10.4 and later.

Declared In

`AuthorizationPlugin.h`

GetArguments

Read the arguments for this authorization mechanism from the authorization policy database. Authorization policy database arguments have not yet been implemented.

```
OSStatus (*GetArguments)(
    AuthorizationEngineRef inEngine,
    const AuthorizationValueVector **outArguments);
```

Parameters

inEngine
: An opaque handle that is passed to your plug-in when the authorization engine calls your MechanismCreate (page 20) function.

outValue
: On input, allocate a pointer to an AuthorizationValueVector structure. On output, the structure contains the number of arguments and a pointer to the data. Because your AuthorizationValueVector structure does not own the data, you must not deallocate the structure or the data pointed to by the structure.

Return Value

A result code. Possible results are errAuthorizationSuccess (no error) and errAuthorizationInternal (Security Server internal error).

Discussion

The authorization policy database might contain arguments for each authentication mechanism. You can use this function to retrieve these arguments.

Important: As of Mac OS X v 10.4, this feature has not been implemented.

The authorization engine sends you the entry point to the GetArguments function in an AuthorizationCallbacks structure when you call the AuthorizationPluginCreate (page 18) function.

Availability

This function is available in Mac OS X v10.4 and later; however, the database does not yet support arguments.

Declared In

AuthorizationPlugin.h

GetContextValue

Read a value collected during authorization.

```
OSStatus (*GetContextValue)(
    AuthorizationEngineRef inEngine,
    AuthorizationString inKey,
    AuthorizationContextFlags *outContextFlags,
    const AuthorizationValue **outValue);
```

Parameters

inEngine
: An opaque handle that is passed to your plug-in when the authorization engine calls your MechanismCreate (page 20) function.

inKey
: A key indicating which value you want to retrieve. This key must correspond to one you specified when you used the SetContextValue (page 15) function to store a context value.

outContextFlags
: On output points to a flag that indicates whether this value is available to the authorization client.

outValue

On input, allocate a pointer to an `AuthorizationValue` structure. On output, the structure contains the size of the data and a pointer to the data. Because your `AuthorizationValue` structure does not own the data, you must not deallocate the structure or the data pointed to by the structure.

Return Value

A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion

Your plug-in authorization mechanism might collect data such as the user name and other authentication information during evaluation of authorization. You can use the `SetContextValue` (page 15) function to have the Security Server store this data and the `GetContextValue` function to retrieve it.

The authorization engine sends you the entry point to the `GetContextValue` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Do not call this function after you have called the `SetResult` (page 17) function. If you do so, the data retrieved by the `GetContextValue` function might not reflect the current value even though the function returns the `errAuthorizationSuccess` result code.

Availability

Available in Mac OS X v10.4 and later.

Declared In

`AuthorizationPlugin.h`

GetHintValue

Read a value stored by the plug-in authorization mechanism.

```
OSStatus (*GetHintValue)(
    AuthorizationEngineRef inEngine,
    AuthorizationString inKey,
    const AuthorizationValue **outValue);
```

Parameters

inEngine

An opaque handle that is passed to your plug-in when the authorization engine calls your `MechanismCreate` (page 20) function.

inKey

A key indicating which value you want to retrieve. This key must correspond to one you specified when you used the `GetHintValue` (page 12) function to store a hint value.

outValue

On input, allocate a pointer to an `AuthorizationValue` structure. On output, the structure contains the size of the data and a pointer to the data. Because your `AuthorizationValue` structure does not own the data, you must not deallocate the structure or the data pointed to by the structure.

Return Value

A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion

Your plug-in authorization mechanism can save and retrieve auxiliary information—called hints—for use by subsequent mechanisms that are part of the same authorization. You use the `SetHintValue` (page 16) function to have the Security Server store this data and the `GetHintValue` function to retrieve it. Hints are not preserved as part of the authorization result; once all mechanisms have approved the authorization or any mechanism has denied it, the security engine disposes of the hints.

The authorization engine sends you the entry point to the `GetHintValue` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Do not call this function after you have called the `SetResult` (page 17) function. If you do so, the data retrieved by the `GetHintValue` function might not reflect the current value even though the function returns the `errAuthorizationSuccess` result code.

Availability

Available in Mac OS X v10.4 and later.

Declared In

`AuthorizationPlugin.h`

GetSessionID

Read the session ID.

```
OSStatus (*GetSessionID)(
    AuthorizationEngineRef inEngine,
    AuthorizationSessionId *outSessionId);
```

Parameters

`inEngine`

An opaque handle that is passed to your plug-in when the authorization engine calls your `MechanismCreate` (page 20) function.

`outSessionId`

On output, points to the session ID.

Return Value

A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion

The session ID is a unique value provided by the authorization engine for a given authorization session. Normally, all the mechanisms in your plug-in are called in turn for a given authorization session and there is no need to ask for the session ID. However, if you were to launch an authorization daemon (for example) that caches data from different authorization sessions and then uses that data later, you might need to keep track of which session a given data item came from. The session ID is available for your use if you wish to implement such a system.

The authorization engine sends you the entry point to the `GetSessionID` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Availability

Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

RequestInterrupt

Request the authorization engine to interrupt the currently active authorization mechanism.

```
OSStatus (*RequestInterrupt)(
    AuthorizationEngineRef inEngine,
);
```

Parameters

`inEngine`
An opaque handle that is passed to your plug-in when the authorization engine calls your `MechanismCreate` (page 20) function.

Return Value
A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion
When you call this function, the security engine calls the `MechanismDeactivate` (page 21) function for your plug-in's currently-active mechanism; that is, the mechanism that was last invoked and that has not yet called the `SetResult` (page 17) function to report its result. Your mechanism should then stop any active processing and call the `DidDeactivate` (page 10) function. When all mechanisms are inactive (that is, they have called either `SetResult` or `DidDeactivate`), the authorization engine calls the `MechanismInvoke` (page 22) function for the mechanism that called `RequestInterrupt` so that it can resume the authorization process from that point. After all mechanisms have called `SetResult`, the authorization engine calls each mechanism's `MechanismDestroy` (page 21) function.

If your mechanism spins off a separate process or UI thread, that thread can call the `RequestInterrupt` function to reinvoke the mechanism, even if that mechanism has already called the `SetResult` (page 17) function. For example, if your plug-in implements a smart card authentication method, reading and evaluating the card might take several minutes to perform. Therefore, in order to avoid blocking other processing while the card is being evaluated, you might spin off a UI thread to interact with the user and then return from `MechanismInvoke` (page 22). When the card has been read, the UI thread calls the `SetResult` function with a value of `kAuthorizationResultAllow` and changes the UI to request the user's PIN. The authorization engine calls the next mechanism, which verifies the PIN. If the user pulls out the card before the verification is complete, the UI thread can call `RequestInterrupt`. The authorization engine then calls the active mechanism's `MechanismDeactivate` (page 21) function, causing it to terminate the PIN verification and call `DidDeactivate` (page 10). Then the authorization engine calls your UI mechanism's `MechanismInvoke` (page 22) function again. Your UI can then prompt the user to reinsert the card.

To understand this sequence better, suppose your plug-in contains three mechanisms: A, B, and C. Mechanism A has called `SetResult` (page 17) and has no active processes. Mechanism B has called `SetResult`, but still has a UI thread running. Mechanism C is running and has not yet called `SetResult`. The user clicks Cancel or otherwise interrupts the UI thread, causing the UI thread to call the `RequestInterrupt` function. The following sequence of events occurs:

1. The authorization engine calls mechanism C's `MechanismDeactivate` (page 21) function.

2. Mechanism C stops active processing and calls the `DidDeactivate` (page 10) function.

3. The authorization engine calls mechanism B's `MechanismInvoke` (page 22) function (because mechanism B is the one that called `RequestInterrupt`).
4. Mechanism B updates the UI and calls the `SetResult` (page 17) function with the value `kAuthorizationResultAllow`.
5. The authorization engine calls mechanism C's `MechanismInvoke` function.
6. Mechanism C completes processing and calls `SetResult` with `kAuthorizationResultAllow`.
7. The authorization engine calls the `MechanismDestroy` (page 21) function of each mechanism in turn (A, B, then C).

The authorization engine sends you the entry point to the `RequestInterrupt` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

SetContextValue

Store data collected during authorization as a key-value pair.

```
OSStatus (*SetContextValue)(
    AuthorizationEngineRef inEngine,
    AuthorizationString inKey,
    AuthorizationContextFlags inContextFlags,
    const AuthorizationValue *inValue);
```

Parameters

`inEngine`
An opaque handle that is passed to your plug-in when the authorization engine calls your `MechanismCreate` (page 20) function.

`inKey`
A key identifying the value you are storing. For standard values such as user names, use the keys listed in `DirectoryService/DirServicesConst.h`. If you need to define a new key, use reverse domain notation (such as `com.apple.ifoo`) and make sure the key is unique. For example, you can use your company name as a prefix for the key name.

`inContextFlags`
A flag that indicates whether this value should be available to the authorization client.

`inValue`
A pointer to an `AuthorizationValue` structure that contains the size of the context data and a pointer to the data. Both the structure and the data are copied to the context maintained by the Security Server.

Return Value
A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion
Your plug-in authorization mechanism collects data such as the user name and other authentication information during evaluation of authorization. You can use this function to have the Security Server store this data and the `GetContextValue` (page 11) function to retrieve it.

When you store this context data, you flag it to indicate whether the authorization client can obtain the value with the `AuthorizationCopyInfo` function. If data is set to be extractable (`kAuthorizationContextFlagExtractable`), it is possible for the authorization client to use the `AuthorizationCopyInfo` function to obtain the value. If data is marked as volatile (`kAuthorizationContextFlagVolatile`), the value is not available to the client. In any case, sensitive data such as a user's password is not provided to the client.

The authorization engine sends you the entry point to the `SetContextValue` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Do not call this function after you have called the `SetResult` (page 17) function. If you do so, the function does not set the context data, even though the function returns the `errAuthorizationSuccess` result code.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

SetHintValue

Store data needed during authorization as a key-value pair.

```
OSStatus (*SetHintValue)(
    AuthorizationEngineRef inEngine,
    AuthorizationString inKey,
    const AuthorizationValue *inValue);
```

Parameters

`inEngine`
An opaque handle that is passed to your plug-in when the authorization engine calls your `MechanismCreate` (page 20) function.

`inKey`
A key identifying the value you are storing. For standard values such as a time stamp, use the keys listed in `DirectoryService/DirServicesConst.h`. If you need to define a new key, make sure the key is unique. For example, you can use your company name as a prefix for the key name.

`inValue`
A pointer to an `AuthorizationValue` structure that contains the size of the data and a pointer to the data. Both the structure and the data are copied to storage maintained by the authorization engine.

Return Value
A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion

Your plug-in authorization mechanism can save and retrieve auxiliary information—called hints—for use by subsequent mechanisms that are part of the same authorization. You use the `SetHintValue` function to have the Security Server store this data and the `GetHintValue` (page 12) function to retrieve it. Hints are not preserved as part of the authorization result; once all mechanisms have approved the authorization or any mechanism has denied it, the security engine disposes of the hints.

The authorization engine sends you the entry point to the `SetHintValue` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Do not call this function after you have called the `SetResult` (page 17) function. If you do so, the function does not set the hint data, even though the function returns the `errAuthorizationSuccess` result code.

Availability

Available in Mac OS X v10.4 and later.

Declared In

`AuthorizationPlugin.h`

SetResult

Return the result of an authorization operation.

```
OSStatus (*SetResult)(
    AuthorizationEngineRef inEngine,
    AuthorizationResult inResult
);
```

Parameters

`inEngine`

An opaque handle that is passed to your plug-in when the authorization engine calls your `MechanismCreate` (page 20) function.

`inResult`

The result of the authorization attempt. See "Authorization Result" (page 28) for possible values.

Return Value

A result code. Possible results are `errAuthorizationSuccess` (no error) and `errAuthorizationInternal` (Security Server internal error).

Discussion

When an application calls the `AuthorizationCopyRights` function to request a specific authorization right, the Security Agent looks for that right in the authorization policy database. If that right corresponds to your plug-in, the authorization engine calls the `MechanismInvoke` (page 22) function for each mechanism listed in the policy database for your plug-in.

When the authorization engine calls your `MechanismInvoke` (page 22) function, your plug-in should invoke the specified mechanism to attempt an authorization operation. You use the `SetResult` function to return the results of this operation. If the mechanism returns `kAuthorizationResultAllow`, then the authorization engine calls the next mechanism (if any) specified in the authorization policy database for the policy. If any of the mechanisms report a result other than `kAuthorizationResultAllow`, the authorization attempt fails. If all of the mechanisms report results of `kAuthorizationResultAllow`, the authorization is considered to have succeeded.

Note that you can spin off a separate process and return from `MechanismInvoke` (page 22) before calling `SetResult`. For example, you might do so to avoid blocking the Security Server if your mechanism takes a significant amount of time to complete or if you want to be able to cancel the operation by calling the `RequestInterrupt` (page 14) function (if, for example, the user has clicked Cancel).In that case, your separate process must call the `SetResult` function to report the result; the authorization engine does not call the next mechanism until you do so.

The authorization engine sends you the entry point to the `SetResult` function in an `AuthorizationCallbacks` structure when you call the `AuthorizationPluginCreate` (page 18) function.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

Callbacks by Task

Initializing a Plug-in

`AuthorizationPluginCreate` (page 18)
Initializes the plug-in and exchanges interfaces with the authorization engine.

Authorization Plug-in Interface Functions

You must declare and implement the functions referred to in the `AuthorizationPluginInterface` structure that you pass to the authorization engine with the `AuthorizationPluginCreate` (page 18) function.

`PluginDestroy` (page 23)
Notifies the plug-in that it is about to be unloaded.

`MechanismCreate` (page 20)
Create an authorization mechanism.

`MechanismInvoke` (page 22)
Invoke an authorization mechanism to perform an authorization operation.

`MechanismDeactivate` (page 21)
Deactivate an authorization mechanism.

`MechanismDestroy` (page 21)
Destroy an authorization mechanism.

Callbacks

AuthorizationPluginCreate

Initializes the plug-in and exchanges interfaces with the authorization engine.

```
OSStatus AuthorizationPluginCreate (
   const AuthorizationCallbacks *callbacks,
   AuthorizationPluginRef *outPlugin,
   const AuthorizationPluginInterface **outPluginInterface
);
```

Parameters

callbacks

A pointer to a structure containing entry points to the Security Server. The functions in this interface are described in "Calling the Authorization Engine" (page 10).

outPlugin

On input, a pointer that you can assign, on output, to a reference value that you define. The authorization engine passes this reference back to you in any subsequent calls to your functions `outPluginInterface->MechanismCreate` (`MechanismCreate` (page 20)) and `outPluginInterface->PluginDestroy` (`MechanismDestroy` (page 21)) so that you can identify the instance of the plug-in affected.

outPluginInterface

On input, a pointer that you assign, on output, to a structure containing entry points in the plug-in. This structure remains valid until the authorization engine calls `outPluginInterface->PluginDestroy`.

Return Value

A result code. Return `errAuthorizationSuccess` (no error) if the function completes successfully and `errAuthorizationInternal` (Security Server internal error) if any error occurs.

Discussion

This function is the main entry point to the plug-in. The authorization engine calls this function only once. The plug-in receives a structure (`AuthorizationCallbacks` (page 26)) containing the entry points to the Security Server's functions (described in "Calling the Authorization Engine" (page 10)) and returns a structure (`AuthorizationPluginInterface` (page 27)) containing the entry points to all of the plug-in's routines ("Authorization Plug-in Interface Functions" (page 18)). Both of these structures contain version numbers. The authorization engine matches the version of its interface to the version in your plug-in's `AuthorizationPluginInterface` structure in order to ensure that older plug-ins will continue to function correctly after the Security Server is updated.

If your plug-in is running in Mac OS X v10.5 or later and displays a window before the user has logged in, you must set the `KHIWindowBitCanBeVisibleWithoutLogin` flag on the window.

For Cocoa, the `NSWindow` method to do this is:

```
- (void)setCanBecomeVisibleWithoutLogin:(BOOL)flag;
```

This method is available in Mac OS X v10.5 and later; see *NSWindow Class Reference*.

For Carbon, you set the `KHIWindowBitCanBeVisibleWithoutLogin` attribute directly; see *Window Manager Reference*. This attribute is also supported by `IBCarbonRuntime` and when archiving a window.

Important: Authorization plug-ins that put up a GUI or otherwise connect to the window server cannot run as privileged. Note that running GUI code as root is a bad idea in general, because GUI code links in many libraries, any of which could contain security vulnerabilities.

Availability

Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

MechanismCreate

Create an authorization mechanism.

```
OSStatus (*MechanismCreate)(
   AuthorizationPluginRef inPlugin,
   AuthorizationEngineRef inEngine,
   AuthorizationMechanismId mechanismId,
   AuthorizationMechanismRef *outMechanism
);
```

You would declare your function like this if you were to name it `MyMechanismCreate`:

```
OSStatus MyMechanismCreate (
   AuthorizationPluginRef inPlugin,
   AuthorizationEngineRef inEngine,
   AuthorizationMechanismId mechanismId,
   AuthorizationMechanismRef *outMechanism
);
```

Parameters

inPlugin
The authorization plug-in reference you assigned to the plug-in in the `AuthorizationPluginCreate` (page 18) function.

inEngine
An opaque handle that you must pass back to the authorization engine when you call one of the engine's callback functions.

mechanismID
The mechanism ID specified in the authorization policy database is passed to the plug-in so that the plug-in can create the appropriate mechanism.

outMechanism
On output, points to an authorization mechanism reference that you define. The authorization engine includes this reference when it calls your plug-in so that you can identify which instance of a mechanism to invoke, deactivate, or destroy.

Return Value
A result code. Return `errAuthorizationSuccess` (no error) if the function completes successfully and `errAuthorizationInternal` (Security Server internal error) if any error occurs.

Discussion
A given authorization plug-in can implement any number of authorization mechanisms, distinguished by their mechanism names in the authorization policy database. For an example, see Listing 1 (page 8).

When the authorization engine calls your `MechanismCreate` function, you should create a mechanism of the type specified by the `mechanismID` parameter and return an authorization mechanism reference. Subsequently, the authorization engine can call your `MechanismInvoke` (page 22) function to perform an authorization, or can direct you to deactivate or destroy the mechanism instance by calling your `MechanismDeactivate` (page 21) or `MechanismDestroy` (page 21) functions.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

MechanismDeactivate

Deactivate an authorization mechanism.

```
OSStatus (*MechanismDeactivate)(
   AuthorizationMechanismRef inMechanism
);
```

You would declare your function like this if you were to name it `MyMechanismDeactivate`:

```
OSStatus MyMechanismDeactivate (
   AuthorizationMechanismRef inMechanism
);
```

Parameters

`inMechanism`
An authorization mechanism reference that you returned when your `MechanismCreate` (page 20) function was called to create the mechanism.

Return Value
A result code. Return `errAuthorizationSuccess` (no error) if the function completes successfully and `errAuthorizationInternal` (Security Server internal error) if any error occurs.

Discussion
The authorization engine calls the `MechanismDeactivate` function of each active mechanism when you call the `RequestInterrupt` (page 14) function. To deactivate your mechanism, you must stop any processing that is currently underway; for example, you should terminate any threads or UI processes that you initiated.

After you have terminated all processing, you must call the `DidDeactivate` (page 10) function; the authorization engine waits for you to call this function before it resumes operation.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

MechanismDestroy

Destroy an authorization mechanism.

```
OSStatus (*MechanismDestroy)(
   AuthorizationMechanismRef inMechanism
);
```

You would declare your function like this if you were to name it `MyMechanismDestroy`:

```
OSStatus MyMechanismDeactivate (
   AuthorizationMechanismRef inMechanism
);
```

Parameters

inMechanism

An authorization mechanism reference that you returned when your `MechanismCreate` (page 20) function was called to create the mechanism.

Return Value

A result code. Return `errAuthorizationSuccess` (no error) if the function completes successfully and `errAuthorizationInternal` (Security Server internal error) if any error occurs.

Discussion

When the authorization engine calls your `MechanismDestroy` function, you must release all resources owned by your mechanism and do any other cleanup necessary (such as deleting temporary files).

Availability

Available in Mac OS X v10.4 and later.

Declared In

`AuthorizationPlugin.h`

MechanismInvoke

Invoke an authorization mechanism to perform an authorization operation.

```
OSStatus (*MechanismInvoke)(
   AuthorizationMechanismRef inMechanism
);
```

You would declare your function like this if you were to name it `MyMechanismInvoke`:

```
OSStatus MyMechanismInvoke (
   AuthorizationMechanismRef inMechanism
);
```

Parameters

inMechanism

An authorization mechanism reference that you returned when your `MechanismCreate` (page 20) function was called to create the mechanism.

Return Value

A result code. Return `errAuthorizationSuccess` (no error) if the function completes successfully and `errAuthorizationInternal` (Security Server internal error) if any error occurs.

Discussion

When the authorization engine calls your `MechanismInvoke` function, you should perform the authorization operation indicated by the mechanism reference. You can use the functions `GetArguments` (page 10), `GetContextValue` (page 11), and `GetHintValue` (page 12) to get more information, if any, about the authorization.

Availability

Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

PluginDestroy

Notifies the plug-in that it is about to be unloaded.

```
OSStatus (*PluginDestroy)(
   AuthorizationPluginRef inPlugin
);
```

You would declare your function like this if you were to name it `MyPluginDestroy`:

```
OSStatus MyPluginDestroy (
   AuthorizationPluginRef inPlugin
);
```

Parameters

`inPlugin`
The authorization plug-in reference you assigned to the plug-in in the `AuthorizationPluginCreate` (page 18) function.

Return Value
A result code. Return `errAuthorizationSuccess` (no error) if the function completes successfully and `errAuthorizationInternal` (Security Server internal error) if any error occurs.

Discussion
When this function is called, your plug-in should release any resources it is holding and do any other cleanup necessary (such as deleting temporary files) before it is unloaded.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

Data Types

AuthorizationValue

Used to pass data between the authorization engine and the plug-in mechanism.

```
typedef struct AuthorizationValue {
    UInt32          length;
    void            *data;
} AuthorizationValue;
```

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationValueVector

Used to pass arguments from the authorization policy database to the authorization mechanism.

```
typedef struct AuthorizationValueVector {
    UInt32              count;
    AuthorizationValue  *values;
} AuthorizationValueVector;
```

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationMechanismID

The mechanism ID specified in the authorization policy database is passed to the plug-in to create the appropriate mechanism.

```
typedef const AuthorizationString AuthorizationMechanismId;
```

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationPluginRef

Handle passed by the plug-in to the authorization engine when the plug-in is initiated.

```
typedef void *AuthorizationPluginRef;
```

Discussion
Your `AuthorizationPluginCreate` (page 18) function assigns this value and returns it to the authorization engine. The authorization engine passes this reference back to you in any subsequent calls to your `MechanismCreate` (page 20) and `PluginDestroy` (page 23) functions.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationMechanismRef

Handle passed by the plug-in to the authorization engine when creating an instance of a mechanism.

```
typedef void *AuthorizationMechanismRef;
```

Discussion
Your `MechanismCreate` (page 20) function assigns this value and returns it to the authorization engine. The authorization engine passes this reference back to you in any subsequent calls to your `MechanismInvoke` (page 22), `MechanismDeactivate` (page 21), and `MechanismDestroy` (page 21) functions.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationEngineRef

Handle passed from the authorization engine to an instance of a mechanism in a plug-in.

```
typedef struct __OpaqueAuthorizationEngine *AuthorizationEngineRef;
```

Discussion
The authorization engine passes one of these opaque handles to your plug-in when it calls your `MechanismCreate` (page 20) function. Your mechanism must pass this handle back to the authorization engine when you call one of the engine's callback functions (see "Calling the Authorization Engine" (page 10)).

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationSessionId

A unique value for an authorization session, provided by the authorization engine.

```
typedef void *AuthorizationSessionId;
```

Discussion
You can call the `GetSessionID` (page 13) function to retrieve the authorization session ID.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationResult

The data type for the result of an authorization evaluation.

```
typedef UInt32 AuthorizationResult;
```

Discussion
The permissible values for an authorization result are enumerated in "Authorization Result" (page 28).

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationCallbacks

The interface implemented by the Security Server.

```
typedef struct AuthorizationCallbacks {
    UInt32 version;
    OSStatus (*SetResult)(
        AuthorizationEngineRef inEngine,
        AuthorizationResult inResult);
    OSStatus (*RequestInterrupt)(
        AuthorizationEngineRef inEngine);
    OSStatus (*DidDeactivate)(
        AuthorizationEngineRef inEngine);
    OSStatus (*GetContextValue)(
        AuthorizationEngineRef inEngine,
        AuthorizationString inKey,
        AuthorizationContextFlags *outContextFlags,
        const AuthorizationValue **outValue);
    OSStatus (*SetContextValue)(
        AuthorizationEngineRef inEngine,
        AuthorizationString inKey,
        AuthorizationContextFlags inContextFlags,
        const AuthorizationValue *inValue);
    OSStatus (*GetHintValue)(
        AuthorizationEngineRef inEngine,
        AuthorizationString inKey,
        const AuthorizationValue **outValue);
    OSStatus (*SetHintValue)(
        AuthorizationEngineRef inEngine,
        AuthorizationString inKey,
        const AuthorizationValue *inValue);
    OSStatus (*GetArguments)(
        AuthorizationEngineRef inEngine,
        const AuthorizationValueVector **outArguments);
    OSStatus (*GetSessionId)(
        AuthorizationEngineRef inEngine,
        AuthorizationSessionId *outSessionId);
} AuthorizationCallbacks;
```

Discussion
This structure is passed to your plug-in through the `AuthorizationPluginCreate` (page 18) function. The functions defined by this structure are described in "Calling the Authorization Engine" (page 10).

Availability
Available in Mac OS X v10.4 and later.

Declared In
`AuthorizationPlugin.h`

AuthorizationPluginInterface

The interface that must be implemented by your plug-in.

```
typedef struct AuthorizationPluginInterface
    UInt32 version;
    OSStatus (*PluginDestroy)(
        AuthorizationPluginRef inPlugin);
    OSStatus (*MechanismCreate)(
        AuthorizationPluginRef inPlugin,
        AuthorizationEngineRef inEngine,
        AuthorizationMechanismId mechanismId,
        AuthorizationMechanismRef *outMechanism);
    OSStatus (*MechanismInvoke)(
        AuthorizationMechanismRef inMechanism);
    OSStatus (*MechanismDeactivate)(
        AuthorizationMechanismRef inMechanism);
    OSStatus (*MechanismDestroy)(
        AuthorizationMechanismRef inMechanism);
} AuthorizationPluginInterface;
```

Discussion
Your plug-in passes this interface to the authorization engine through the `AuthorizationPluginCreate` (page 18) function. The functions defined by this structure are described in "Functions Implemented By the Plug-in" (page 18).

Constants

Authorization Context Flags

Defines flags that specify whether authentication data should be made available to the authorization client.

```
typedef UInt32 AuthorizationContextFlags;
enum {
    kAuthorizationContextFlagExtractable =  (1 << 0),
    kAuthorizationContextFlagVolatile =     (1 << 1),
    kAuthorizationContextFlagSticky =       (1 << 2)
};
```

Constants
`kAuthorizationContextFlagExtractable`

It is possible for the authorization client to use the `AuthorizationCopyInfo` function to obtain the value.

Available in Mac OS X v10.4 and later.

Declared in `AuthorizationPlugin.h`.

`kAuthorizationContextFlagVolatile`

The value is not saved for the authorization client.

Available in Mac OS X v10.4 and later.

Declared in `AuthorizationPlugin.h`.

`kAuthorizationContextFlagSticky`

This data persists through an interrupted or failed evaluation.

This flag can be used to propagate an error condition from a downstream plug-in to an upstream one. It is not remembered in the authorization reference (see *Authorization Services C Reference*.

Available in Mac OS X v10.5 and later.

Declared in `AuthorizationPlugin.h`.

Authorization Result

The result of an authorization evaluation.

```
enum {
    kAuthorizationResultAllow,
    kAuthorizationResultDeny,
    kAuthorizationResultUndefined,
    kAuthorizationResultUserCanceled,
};
```

Constants

`kAuthorizationResultAllow`

The authorization operation succeeded and authorization should be granted.

Available in Mac OS X v10.4 and later.

Declared in `AuthorizationPlugin.h`.

`kAuthorizationResultDeny`

The authorization operation succeeded and authorization should be denied.

Available in Mac OS X v10.4 and later.

Declared in `AuthorizationPlugin.h`.

`kAuthorizationResultUndefined`

The authorization operation failed and should not be retried for this session.

Available in Mac OS X v10.4 and later.

Declared in `AuthorizationPlugin.h`.

`kAuthorizationResultUserCanceled`

The user has requested that the authorization evaluation be terminated.

Available in Mac OS X v10.4 and later.

Declared in `AuthorizationPlugin.h`.

Plug-in Interface Version

The version of the interface implemented by the plug-in.

```
enum {
    kAuthorizationPluginInterfaceVersion =  0
};
```

Discussion
The plug-in interface is defined by the `AuthorizationPluginInterface` (page 27) structure and described in "Functions Implemented By the Plug-in" (page 18).

Authorization Engine Interface Version

The version of the interface implemented by the authorization engine.

```
enum {
    kAuthorizationCallbacksVersion =        0
};
```

Discussion
The authorization engine interface is defined by the `AuthorizationCallbacks` (page 26) structure and described in "Calling the Authorization Engine" (page 10).

Result Codes

The result codes used by authorization plug-ins are listed in the table below.

Result Code	Value	Description
`errAuthorizationSuccess`	0	The operation completed successfully. Available in Mac OS X v10.0 and later.
`errAuthorizationInternal`	-60008	An unrecognized internal error occurred. Available in Mac OS X v10.0 and later.

Document Revision History

This table describes the changes to *Authorization Plug-in Reference*.

Date	Notes
2007-05-15	Added the Sticky authorization context flag to the Constants section; added a note about tagging windows that can be visible before login.
2005-04-29	New document that describes the interface for creating plug-ins that can participate in authorization decisions.
	New document that describes the interface for creating plug-ins that can participate in authorization decisions.

Index