App Sandbox Design Guide

Developer

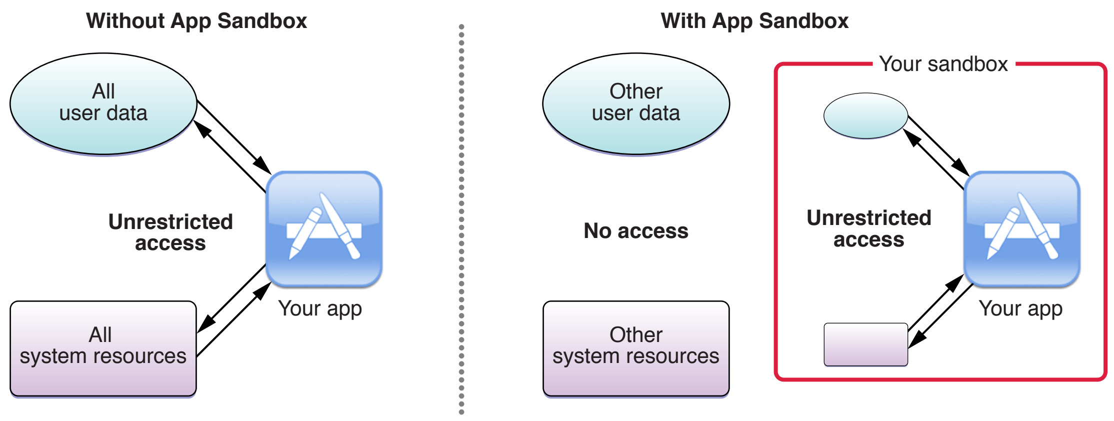
Contents

Tables and Listings

About App Sandbox

A non-sandboxed app has the full rights of the user who is running that app, and can access any resources that the user can access. If that app or the frameworks it is linked against contain security holes, an attacker can potentially exploit those holes to take control of that app, and in doing so, the attacker gains the ability to do anything that the user can do.

By limiting access to resources on a per-app basis, App Sandbox provides a last line of defense against the theft, corruption, or deletion of user data if an attacker successfully exploits security holes in your app or the frameworks it is linked against.



App Sandbox is an access control technology provided in OS X, enforced at the kernel level. Its strategy is twofold:

1. App Sandbox enables you to describe how your app interacts with the system. The system then grants your app the access it needs to get its job done, and no more.
2. App Sandbox allows the user to transparently grant your app additional access by way of Open and Save dialogs, drag and drop, and other familiar user interactions.

At a Glance

Based on simple security principles, App Sandbox provides strong defense against damage from malicious code. The elements of App Sandbox are container directories, entitlements, user-determined permissions, privilege separation, and kernel enforcement. It's up to you to understand these elements and then to use your understanding to create a plan for adopting App Sandbox.

Relevant chapters: App Sandbox Quick Start (page 8), App Sandbox in Depth (page 14)

After you understand the basics, look at your app in light of this security technology. First, determine if your app is suitable for sandboxing. (Most apps are.) Then resolve any API incompatibilities and determine which entitlements you need. Finally, consider applying privilege separation to maximize the defensive value of App Sandbox.

Relevant chapter: Designing for App Sandbox (page 32)

Some file system locations that your app uses are different when you adopt App Sandbox. In particular, you gain a container directory to be used for app support files, databases, caches, and other files apart from user documents. OS X and Xcode support migration of files from their legacy locations to your container.

Relevant chapter: Migrating an App to a Sandbox (page 39)

How to Use This Document

To get up and running with App Sandbox, perform the tutorial in App Sandbox Quick Start (page 8). Before sandboxing an app you intend to distribute, be sure you understand App Sandbox in Depth (page 14). When you're ready to start sandboxing a new app, or to convert an existing app to adopt App Sandbox, read Designing for App Sandbox (page 32). If you're providing a new, sandboxed version of your app to users already running a version that is not sandboxed, read Migrating an App to a Sandbox (page 39).

Prerequisites

Before you read this document, make sure you understand the place of App Sandbox and code signing in the overall OS X development process by reading *Mac App Programming Guide*.

See Also

To complement the damage containment provided by App Sandbox, you must provide a first line of defense by adopting secure coding practices throughout your app. To learn how, read *Security Overview* and *Secure Coding Guide*.

An important step in adopting App Sandbox is requesting entitlements for your app. For details on all the available entitlements, see *Entitlement Key Reference*.

You can enhance the benefits of App Sandbox in a full-featured app by implementing privilege separation. You do this using XPC, an OS X implementation of interprocess communication. To learn the details of using XPC, read *Daemons and Services Programming Guide*.

App Sandbox Quick Start

In this Quick Start you get an OS X app up and running in a sandbox. You verify that the app is indeed sandboxed and then learn how to troubleshoot and resolve a typical App Sandbox error. The apps you use are Xcode, Keychain Access, Activity Monitor, and Console.

Create the Xcode Project

The app you create in this Quick Start uses a WebKit web view and consequently uses a network connection. Under App Sandbox, network connections don't work unless you specifically allow them—making this a good example app for learning about sandboxing.

To create the Xcode project for this Quick Start

1. In Xcode 4, create a new Xcode project for an OS X Cocoa application.
    - Name the project AppSandboxQuickStart.
    - Set a company identifier, such as `com.yourcompany`, if none is already set.
    - Ensure that Use Automatic Reference Counting is selected and that the other checkboxes are unselected.

2. In the project navigator, click the MainMenu nib file (`MainMenu.xib`).

   The Interface Builder canvas appears.

3. In the Xcode dock, click the Window object.

   The app's window is now visible on the canvas.

4. In the object library (in the utilities area), locate the `WebView` object.
5. Drag a web view onto the window on the canvas.
6. (Optional) To improve the display of the web view in the running app, perform the following steps:
    - Drag the sizing controls on the web view so that it completely fills the window's main view.
    - Using the size inspector for the web view, ensure that all of the inner and outer autosizing constraints are active.

7. Add the WebKit framework to the app.
    - Import the WebKit framework by adding the following statement above the interface block in the `AppDelegate.h` header file:

    ```
    #import <WebKit/WebKit.h>
    ```

    - Link the WebKit framework to the Quick Start project as a required framework.

> **Note:** If you are compiling for OS X v10.7 and want to play HTML5 embedded videos in this application, you must also link to the AV Foundation framework. This is not required in OS X v10.8 and later.

8. Create and connect an outlet for the web view in the `AppDelegate` class.

    In the app delegate's interface (either in `AppDelegate.h` or in a category in `AppDelegate.m`), add this:

    ```
    @property (weak) IBOutlet WebView *webView;
    ```

    Then synthesize the property in the implementation (in `AppDelegate.m`):

    ```
    @synthesize webView = _webView;
    ```

9. Connect the web view to the app delegate outlet you just created.
10. Add the following `awakeFromNib` method to the `AppDelegate.m` implementation file:

    ```
    - (void) awakeFromNib {
        [self.webView.mainFrame loadRequest:
            [NSURLRequest requestWithURL:
                [NSURL URLWithString: @"http://www.apple.com"]]];
    }
    ```

    On application launch, this method requests the specified URL from the computer's network connection and then sends the result to the web view for display.

Now, build and run the app—which is not yet sandboxed and so has free access to system resources including its network sockets. Confirm that the app's window displays the page you specified in the `awakeFromNib` method. (If you get a blank window, make sure the outlet is properly connected to the web view.)

When done, quit the app.

Enable App Sandbox

Important: For Xcode 6 and later, read *App Distribution Quick Start* for how to create your team provisioning profile and enable App Sandbox.

You enable App Sandbox by selecting a checkbox in the Xcode target editor.

In Xcode, click the project file in the project navigator and click the `AppSandboxQuickStart` target, if they're not already selected. View the Summary tab of the target editor.

To enable App Sandbox for the project

1. In the Summary tab of the target editor, be sure the "Use Entitlements file" checkbox is checked, and specify a file for those entitlements.

   An *entitlement* is a key-value pair, defined in a property list file, that confers a specific capability or security permission to a target.

   When you click Enable Entitlements, Xcode automatically checks the Code Sign Application checkbox and the Enable App Sandboxing checkbox. Together, these are the essential project settings for enabling App Sandbox.

   When you click "Use Entitlements file", Xcode also creates a `.entitlements` property list file, visible in the project navigator. As you use the graphical entitlements interface in the target editor, Xcode updates the property list file.

2. If you are using an older version of Xcode that does not have a checkbox for enabling iCloud, clear the contents of the iCloud entitlement fields.

   This Quick Start doesn't use iCloud. Because older versions of Xcode automatically added iCloud entitlement values when you enabled entitlements, delete them as follows:

   - In the Summary tab of the target editor, select and then delete the content of the iCloud Key-Value Store field.
   - Click the top row in the iCloud Containers field and click the minus button.

At this point in the Quick Start, you have enabled App Sandbox but have not yet provided a code signing identity for the Xcode project. If you have already configured your system with a signing identity, Xcode should default to using your development identity when signing this app.

If your system is not configured with a development identity and you attempt to build the project, the build will fail; before you continue with this tutorial, create a code signing identity as described in *App Distribution Guide* .

Now, build the app.

Confirm That the App Is Sandboxed

Build and run the Quick Start app. The window opens, but if the app is successfully sandboxed, no web content appears. This is because you have not yet conferred permission to access a network connection.

Apart from blocked behavior, there are three specific signs that an OS X app is successfully sandboxed.

To confirm that the Quick Start app is successfully sandboxed

1. In Finder, look at the contents of the `~/Library/Containers/` folder.

   If the Quick Start app is sandboxed, there is now a container folder named after your app. The name includes the company identifier for the project, so the complete folder name would be, for example, `com.yourcompany.AppSandboxQuickStart`.

   The system creates an app's container folder, for a given user, the first time the user runs the app.

2. In Activity Monitor, check that the system recognizes the app as sandboxed.
   - Launch Activity Monitor (available in `/Applications/Utilities`).
   - In Activity Monitor, choose View > Columns.

     Ensure that the Sandbox menu item is checked.
   - In the Sandbox column, confirm that the value for the Quick Start app is Yes.

     To make it easier to locate the app in Activity monitor, enter the name of the Quick Start app in the Filter field.

3. Check that the app binary is sandboxed.

```
codesign -dvvv --entitlements :- executable_path
```

where `executable_path` is the complete path for the app's main executable binary (for example, `Quick Start.app/Contents/MacOS/Quick Start`).

Important: The above steps are sufficient for the Quick Start app, but are not sufficient for apps that contain embedded helper apps, XPC services, or other tools. For more information, read External Tools, XPC Services, and Privilege Separation (page 28).

Tip: If the app crashes when you attempt to run it, specifically by receiving an `EXC_BAD_INSTRUCTION` signal, the most likely reason is that you previously ran a sandboxed app with the same bundle identifier but a different code signature. This crashing upon launch is an App Sandbox security feature that prevents one app from masquerading as another and thereby gaining access to the other app's container.

You learn how to design and build your apps, in light of this security feature, in App Sandbox and Code Signing (page 26).

Resolve an App Sandbox Violation

An App Sandbox violation occurs if your app tries to do something that App Sandbox does not allow. For example, you have already seen in this Quick Start that the sandboxed app is unable to retrieve content from the web. Fine-grained restriction over access to system resources is the heart of how App Sandbox provides protection should an app become compromised by malicious code.

The most common source of App Sandbox violations is a mismatch between the entitlement settings you specified in Xcode and the needs of your app. In this section you observe and then correct an App Sandbox violation.

To diagnose an App Sandbox violation

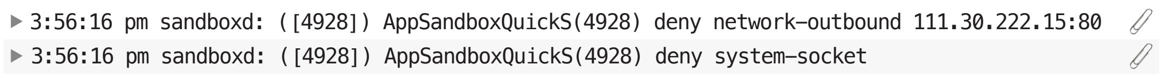
1. Build and run the Quick Start app.

   The app starts normally, but fails to display the webpage specified in its `awakeFromNib` method (as you've previously observed in Confirm That the App Is Sandboxed (page 11)). Because displaying the webpage worked correctly before you sandboxed the app, it is appropriate in this case to suspect an App Sandbox violation.

2. Open the Console application (available in `/Applications/Utilities/`) and ensure that All Messages is selected in the sidebar.

   In the filter field of the Console window, enter `sandboxd` to display only App Sandbox violations.

`sandboxd` is the name of the App Sandbox daemon that reports on sandbox violations. The relevant messages, as displayed in Console, look similar to the following:



```
▶ 3:56:16 pm sandboxd: ([4928]) AppSandboxQuickS(4928) deny network-outbound 111.30.222.15:80
▶ 3:56:16 pm sandboxd: ([4928]) AppSandboxQuickS(4928) deny system-socket
```

The problem that generates these console messages is that the Quick Start app does not yet have the entitlement for outbound network access.

Tip: To see the full backtraces for either violation, click the paperclip icon near the right edge of the corresponding Console message.

The steps in the previous task illustrate the general pattern to use for identifying App Sandbox violations:

1. Confirm that the violation occurs only with App Sandbox enabled in your project.
2. Provoke the violation (such as by attempting to use a network connection, if your app is designed to do that).
3. Look in Console for `sandboxd` messages.

There is also a simple, general pattern to use for resolving such violations.

To resolve the App Sandbox violation by adding the appropriate entitlement

1. Quit the Quick Start app.
2. In the Summary tab of the target editor, look for the entitlement that corresponds to the reported `sandboxd` violation.

   In this case, the primary error is `deny network-outbound`. The corresponding entitlement is Allow Outgoing Network Connections.

3. In the Summary tab of the target editor, select the Allow Outgoing Network Connections checkbox.

   Doing so applies a `TRUE` value, for the needed entitlement, to the Xcode project.

4. Build and run the app.

   The intended webpage now displays in the app. In addition, there are no new App Sandbox violation messages in Console.

App Sandbox in Depth

The access control mechanisms used by App Sandbox to protect user data are small in number and easy to understand. But the specific steps for you to take, as you adopt App Sandbox, are unique to your app. To determine what those steps are, you must understand the key concepts for this technology.

The Need for a Last Line of Defense

You secure your app against attack from malware by following the practices recommended in *Secure Coding Guide*. But despite your best efforts to build an invulnerable barrier—by avoiding buffer overflows and other memory corruptions, preventing exposure of user data, and eliminating other vulnerabilities—your app can be exploited by malicious code. An attacker needs only to find a single hole in your defenses, or in any of the frameworks and libraries that you link against, to gain control of your app's interactions with the system.

App Sandbox is designed to confront this scenario head on by letting you describe your app's intended interactions with the system. The system then grants your app only the access your app needs to get its job done. If malicious code gains control of a properly sandboxed app, it is left with access to only the files and resources in the app's sandbox.

To successfully adopt App Sandbox, use a different mindset than you might be accustomed to, as suggested in Table 2-1.

Table 2-1 The App Sandbox mindset

When developing…	When adopting App Sandbox…
Add features	Minimize system resource use
Take advantage of access throughout your app	Partition functionality, then distrust each part
Use the most convenient API	Use the most secure API
View restrictions as limitations	View restrictions as safeguards

When designing for App Sandbox, you are planning for the following worst-case scenario: Despite your best efforts, malicious code breaches an unintended security hole—either in your code or in a framework you've linked against. Capabilities you've added to your app become capabilities of the hostile code. Keep this in mind as you read the rest of this document.

Entitlements and System Resource Access

An app that is not sandboxed has access to all user-accessible system resources—including the built-in camera and microphone, network sockets, printing, and most of the file system. If successfully attacked by malicious code, such an app can behave as a hostile agent with wide-ranging potential to inflict harm.

When you enable App Sandbox for your app, you remove all but a minimal set of privileges and then deliberately restore them, one-by-one, using entitlements. An *entitlement* is a key-value pair that identifies a specific capability, such as the capability to open an outbound network socket.

One special entitlement—Enable App Sandboxing—turns on App Sandbox. When you enable sandboxing, Xcode creates a `.entitlements` property list file and shows it in the project navigator.

If your app requires a capability, request it by adding the corresponding entitlement to your Xcode project using the Summary tab of the target editor. If you don't require a capability, take care to not include the corresponding entitlement.

You request entitlements on a target-by-target basis. If your app has a single target—the main application—you request entitlements only for that target. If you design your app to use a main application along with helpers (in the form of XPC services), you request entitlements individually, and as appropriate, for each target. You learn more about this in External Tools, XPC Services, and Privilege Separation (page 28).

You may require finer-grained control over your app's entitlements than is available in the Xcode target editor. For example, you might request a temporary exception entitlement because App Sandbox does not support a capability your app needs, such as the ability to send an Apple event to an app that does not yet provide any scripting access groups. To work with temporary exception entitlements, use the Xcode property list editor to edit a target's `.entitlements` property list file directly.

Note: If you request a temporary exception entitlement, be sure to follow the guidance regarding entitlements provided on the iTunes Connect website. In particular, file a bug asking for the functionality that you need, and use the Review Notes field in iTunes Connect to explain why your app needs the temporary exception. Be sure to provide the bug number.

OS X App Sandbox entitlements are described in Enabling App Sandbox in *Entitlement Key Reference* . For a walk-through of requesting an entitlement for a target in an Xcode project, see App Sandbox Quick Start (page 8).

Container Directories and File System Access

When you adopt App Sandbox, your application has access to the following locations:

- **The app container directory.** Upon first launch, the operating system creates a special directory for use by your app—and only by your app—called a *container* . Each user on a system gets an individual container for your app, within their home directory; your app has unfettered read/write access to the container for the user who ran it.
- **App group container directories.** A sandboxed app can specify an entitlement that gives it access to one or more app group container directories, each of which is shared among all apps with that entitlement.
- **User-specified files.** A sandboxed app (with an appropriate entitlement) automatically obtains access to files in arbitrary locations when those files are explicitly opened by the user or are dragged and dropped onto the application by the user.
- **Related items.** With the appropriate entitlement, your app can access a file with the same name as a user-specified file, but a different extension. This can be used for accessing files that are functionally related (such as a subtitle file associated with a movie) or for saving modified files in a different format (such as re-saving an RTF flat file as an RTFD container after the user added a picture).
- **Temporary directories, command-line tool directories, and specific world-readable locations.** A sandboxed app has varying degrees of access to files in certain other well-defined locations.

These policies are detailed further in the sections that follow.

The App Sandbox Container Directory

The app sandbox container directory has the following characteristics:

- It is located at a system-defined path, within the user's home directory. In a sandboxed app, this path is returned when your app calls the `NSHomeDirectory` function.

- Your app has unrestricted read/write access to the container and its subdirectories.
- OS X path-finding APIs (above the POSIX layer) refer to locations that are specific to your app.

  Most of these path-finding APIs refer to locations relative to your app's container. For example, the container includes an individual `Library` directory (specified by the `NSLibraryDirectory` search path constant) for use only by your app, with individual `Application Support` and `Preferences` subdirectories.

  Using your container for support files requires no code change (from the pre-sandbox version of your app) but may require one-time migration, as explained in Migrating an App to a Sandbox (page 39).

  Some path-finding APIs (above the POSIX layer) refer to app-specific locations outside of the user's home directory. In a sandboxed app, for example, the `NSTemporaryDirectory` function provides a path to a directory that is outside of the user's home directory but specific to your app and within your sandbox; you have unrestricted read/write access to it for the current user. The behavior of these path-finding APIs is suitably adjusted for App Sandbox and no code change is needed.
- OS X establishes and enforces the connection between your app and its container by way of your app's code signature.
- The container is in a hidden location, and so users do not interact with it directly. Specifically, the container is not for user documents. It is for files that your app uses, along with databases, caches, and other app-specific data.

  For a shoebox-style app, in which you provide the only user interface to the user's content, that content goes in the container and your app has full access to it.

> **iOS Note:** Because it is not for user documents, an OS X container differs from an iOS container—which, in iOS, is the one and only location for user documents.
>
> In addition, an iOS container contains the app itself. This is not so in OS X.

> **iCloud Note:** Apple's iCloud technology, as described in *iCloud Design Guide*, uses the name "container" as well. There is no functional connection between an iCloud container and an App Sandbox container.

Thanks to code signing, no other sandboxed app can gain access to your container, even if it attempts to masquerade as your app by using your bundle identifier. Future versions of your app, however—provided that you use the same code signature and bundle identifier—do reuse your app's container.

For each user, a sandboxed app's container directory is created automatically when that user first runs the app. Because a container is within a user's home directory, each user on a system gets their own container for your app.

The Application Group Container Directory

In addition to per-app containers, in OS X v10.7.5 and in OS X v10.8.3 and later, an application can use the `com.apple.security.application-groups` entitlement to request access to a shared container that is common to multiple applications produced by the same development team. This container is intended for content that is not user-facing, such as shared caches or databases.

> **Note:** Applications that are members of an application group also gain the ability to share Mach and POSIX semaphores and to use certain other IPC mechanisms in conjunction with other group members. See IPC and POSIX Semaphores and Shared Memory (page 31) for more details.

These group containers are automatically added into each app's sandbox container as determined by the existence of these keys, and are stored in `~/Library/Group Containers/<application-group-id>`, where `<application-group-id>` is the name of the group. The group name itself must begin with your development team ID, followed by a period.

Beginning in OS X v10.8.3, your app can obtain the path to the group containers by calling the `containerURLForSecurityApplicationGroupIdentifier:` method of `NSFileManager`.

Note: In OS X v10.9, calling this method creates the group container directory automatically, along with `Library/Preferences`, `Library/Caches`, and `Library/Application Support` folders within that group container directory.

In previous versions of OS X, although the group container directory is part of your sandbox, the directory itself is not created automatically. Your app must create this directory as shown in Listing 2-1:

Listing 2-1 Creating an app group container directory

```
    NSFileManager *fm = [NSFileManager defaultManager];
   NSString *appGroupName = @"Z123456789.com.example.app-group"; /* For example
 */


    NSURL *groupContainerURL = [fm
containerURLForSecurityApplicationGroupIdentifier:appGroupName];
    NSError* theError = nil;
    if (![fm createDirectoryAtURL: groupContainerURL
withIntermediateDirectories:YES attributes:nil error:&theError]) {
        // Handle the error.
    }
```

You should organize the contents of this directory in the same way that any other `Library` folder is organized, using standard folder names—`Preferences`, `Application Support`, and so on—as needed.

For more details, see Adding an Application to an Application Group in *Entitlement Key Reference*.

Powerbox and File System Access Outside of Your Container

Your sandboxed app can access file system locations outside of its container in the following three ways:

- At the specific direction of the user
- By using entitlements for specific file-system locations (described in Entitlements and System Resource Access (page 15))
- When the file system location is in certain directories that are world readable

The OS X security technology that interacts with the user to expand your sandbox is called *Powerbox*. Powerbox has no API. Your app uses Powerbox transparently when you use the `NSOpenPanel` and `NSSavePanel` classes. You enable Powerbox by setting an entitlement using Xcode, as described in Enabling User-Selected File Access in *Entitlement Key Reference* .

When you invoke an Open or Save dialog from your sandboxed app, the window that appears is presented not by AppKit but by Powerbox. Using Powerbox is automatic when you adopt App Sandbox—it requires no code change from the pre-sandbox version of your app. Accessory panels that you've implemented for opening or saving are faithfully rendered and used.

Note: When you adopt App Sandbox, there are some important behavioral differences for the `NSOpenPanel` and `NSSavePanel` classes, described in Open and Save Dialog Behavior with App Sandbox (page 22).

The security benefit provided by Powerbox is that it cannot be manipulated programmatically—specifically, there is no mechanism for hostile code to use Powerbox for accessing the file system. Only a user, by interacting with Open and Save dialogs via Powerbox, can use those dialogs to reach portions of the file system outside of your previously established sandbox. For example, if a user saves a new document, Powerbox expands your sandbox to give your app read/write access to the document.

When a user of your app specifies they want to use a file or a folder, the system adds the associated path to your app's sandbox. Say, for example, a user drags the `~/Documents` folder onto your app's Dock tile (or onto your app's Finder icon, or into an open window of your app), thereby indicating they want to use that folder. In response, the system makes the `~/Documents` folder, its contents, and its subfolders available to your app.

If a user instead opens a specific file, or saves to a new file, the system makes the specified file, and that file alone, available to your app.

In addition, the system automatically permits a sandboxed app to:

- Connect to system input methods
- Invoke services chosen by the user from the Services menu (only those services flagged as "safe" by the service provider are available to a sandboxed app)
- Open files chosen by the user from the Open Recent menu
- Participate with other apps by way of user-invoked copy and paste
- Read files that are world readable, in certain directories, including the following directories:
    - /bin
    - /sbin
    - /usr/bin

  - /usr/lib
  - /usr/sbin
  - /usr/share
  - /System
- Read and write files in directories created by calling `NSTemporaryDirectory`.

> **Note:** The `/tmp` directory is *not* accessible from sandboxed apps. You must use the `NSTemporaryDirectory` function to obtain a temporary location for your app's temporary files.

After a user has specified a file they want to use, that file is within your app's sandbox. The file is then vulnerable to attack if your app is exploited by malicious code: App Sandbox provides no protection. To provide protection for the files within your sandbox, follow the recommendations in *Secure Coding Guide*.

By default, files opened or saved by the user remain within your sandbox until your app terminates, except for files that were open *at the time* that your app terminates. Such files reopen automatically by way of the OS X Resume feature the next time your app launches, and are automatically added back to your app's sandbox.

To provide persistent access to resources located outside of your container, in a way that doesn't depend on Resume, use security-scoped bookmarks as explained in Security-Scoped Bookmarks and Persistent Resource Access (page 23).

Related Items

The related items feature of App Sandbox lets your app access files that have the same name as a user-chosen file, but a different extension. This feature consists of two parts: a list of related extensions in the application's `Info.plist` file and code to tell the sandbox what you're doing.

There are two common scenarios where this makes sense:

Scenario 1:

Your app needs to be able to save a file with a different extension than that of the original file. For example, when you paste an image into an RTF file in TextEdit and save it, TextEdit changes the file's extension from `.rtf` to `.rtfd` (and it becomes a directory).

To handle this situation, you must use an `NSFileCoordinator` object to coordinate access to the file. Before you rename the file, call the `itemAtURL:willMoveToURL:` method. After you rename the file, call the `itemAtURL:didMoveToURL:` method.

Scenario 2:

Your app needs to be able to open or save multiple related files with the same name and different extensions (for example, to automatically open a subtitle file with the same name as a movie file, or to allow for a SQLite journal file).

To gain access to that secondary file, create a class that conforms to the `NSFilePresenter` protocol. This object should provide the main file's URL as its `primaryPresentedItemURL` property, and should provide the secondary file's URL as its `presentedItemURL` property.

After the user opens the main file, your file presenter object should call the `addFilePresenter:` class method on the `NSFileCoordinator` class to register itself.

Note: In the case of a SQLite journal file, beginning in 10.8.2, journal files, write-ahead logging files, and shared memory files are automatically added to the related items list if you open a SQLite database, so this step is unnecessary.

In both scenarios, you must make a small change to the application's `Info.plist` file. Your app should already declare a Document Types (`CFBundleDocumentTypes`) array that declares the file types your app can open.

For each file type dictionary in that array, if that file type should be treated as a potentially related type for open and save purposes, add the key `NSIsRelatedItemType` with a boolean value of `YES`.

To learn more about file presenters and file coordinators, read *File System Programming Guide* .

Open and Save Dialog Behavior with App Sandbox

Certain `NSOpenPanel` and `NSSavePanel` methods behave differently when App Sandbox is enabled for your app:

- You cannot invoke the OK button using the `ok:` method.
- You cannot rewrite the user's selection using the `panel:userEnteredFilename:confirmed:` method from the `NSOpenSavePanelDelegate` protocol.

In addition, the effective, runtime inheritance path for the `NSOpenPanel` and `NSSavePanel` classes is different with App Sandbox, as illustrated in Table 2-2.

Table 2-2 Open and Save class inheritance with App Sandbox

Without App Sandbox	NSOpenPanel : NSSavePanel : NSPanel : NSWindow : NSResponder : NSObject
With App Sandbox	NSOpenPanel : NSSavePanel : NSObject

Because of this runtime difference, an `NSOpenPanel` or `NSSavePanel` object inherits fewer methods with App Sandbox. If you attempt to send a message to an `NSOpenPanel` or `NSSavePanel` object, and that method is defined in the `NSPanel`, `NSWindow`, or `NSResponder` classes, the system raises an exception. The Xcode compiler does *not* issue a warning or error to alert you to this runtime behavior.

Security-Scoped Bookmarks and Persistent Resource Access

Your app's access to file-system locations outside of its container—as granted to your app by way of user intent, such as through Powerbox—does not automatically persist across app launches or system restarts. When your app reopens, you have to start over. (The one exception to this is for files open *at the time* that your app terminates, which remain in your sandbox thanks to the OS X Resume feature).

Starting in OS X v10.7.3, you can retain access to file-system resources by employing a security mechanism, known as *security-scoped bookmarks*, that preserves user intent. Here are a few examples of app features that can benefit from this:

- A user-selected download, processing, or output folder
- An image browser library file, which points to user-specified images at arbitrary locations
- A complex document format that supports embedded media stored in other locations

Two Distinct Types of Security-Scoped Bookmark

Security-scoped bookmarks, available starting in OS X v10.7.3, support two distinct use cases:

- An *app-scoped bookmark* provides your sandboxed app with persistent access to a user-specified file or folder.

  For example, if your app employs a download or processing folder that is outside of the app container, obtain initial access by presenting an `NSOpenPanel` dialog to obtain the user's intent to use a specific folder. Then, create an app-scoped bookmark for that folder and store it as part of the app's configuration (perhaps in a property list file or using the `NSUserDefaults` class). With the app-scoped bookmark, your app can obtain future access to the folder.

- A *document-scoped bookmark* provides a specific document with persistent access to a file.

  For example, a video editing app typically supports the notion of a project document that refers to other files and needs persistent access to those files. Such a project document can store security-scoped bookmarks to the files it refers to.

  Obtain initial access to a referred item by asking for user intent to use that item. Then, create a document-scoped bookmark for the item and store the bookmark as part of the document's data.

A document-scoped bookmark can be resolved by any app that has access to the bookmark data itself and to the document that owns the bookmark. This supports portability, allowing a user, for example, to send a document to another user; the document's secure bookmarks remain usable for the recipient. The document can be a single flat file or a package containing multiple files.

A document-scoped bookmark can point only to a file, not a folder, and only to a file that is not in a location used by the system (such as `/private` or `/Library`).

Using Security-Scoped Bookmarks

To use either type of security-scoped bookmark requires you to perform five steps:

1. Set the appropriate entitlement in the target that needs to use security-scoped bookmarks.

   Do this once per target as part of configuring your Xcode project.

2. Create a security-scoped bookmark.

   Do this when a user has indicated intent (such as via Powerbox) to use a file-system resource outside of your app's container, and you want to preserve your app's ability to access the resource.

3. Resolve the security-scoped bookmark.

   Do this when your app later (for example, after app relaunch) needs access to a resource you bookmarked in step 2. The result of this step is a security-scoped URL.

4. Explicitly indicate that you want to use the file-system resource whose URL you obtained in step 3.

   Do this immediately after obtaining the security-scoped URL (or, when you later want to regain access to the resource after having relinquished your access to it).

5. When done using the resource, explicitly indicate that you want to stop using it.

   Do this as soon as you know that you no longer need access to the resource (typically, after you close it).

   After you relinquish access to a file-system resource, to use that resource again you must return to step 4 (to again indicate you want to use the resource).

   If your app is relaunched, you must return to step 3 (to resolve the security-scoped bookmark).

The first step in the preceding list, requesting entitlements, is the prerequisite for using either type of security-scoped bookmark. Perform this step as follows:

- To use app-scoped bookmarks in a target, set the `com.apple.security.files.bookmarks.app-scope` entitlement value to `true`.
- To use document-scoped bookmarks in a target, set the `com.apple.security.files.bookmarks.document-scope` entitlement value to `true`.

You can request either or both of these entitlements in a target, as needed. These entitlements are available starting in OS X v10.7.3 and are described in Enabling Security-Scoped Bookmark and URL Access.

With the appropriate entitlements, you can create a security-scoped bookmark by calling the `bookmarkDataWithOptions:includingResourceValuesForKeys:relativeToURL:error:` method of the `NSURL` class.

When you later need access to a bookmarked resource, resolve its security-scoped bookmark by calling the `URLByResolvingBookmarkData:options:relativeToURL:bookmarkDataIsStale:error:` method of the `NSURL` class.

In a sandboxed app, you cannot access the file-system resource that a security-scoped URL points to until you call the `startAccessingSecurityScopedResource` method on the URL.

When you no longer need access to a resource that you obtained using security scope (typically, after you close the resource) you must call the `stopAccessingSecurityScopedResource` method on the resource's URL.

Calls to start and stop access are not nested. When you call the `stopAccessingSecurityScopedResource` method, you immediately lose access to the resource. If you call this method on a URL whose referenced resource you do not have access to, nothing happens.

Warning: If you fail to relinquish your access to file-system resources when you no longer need them, your app leaks kernel resources. If sufficient kernel resources are leaked, your app loses its ability to add file-system locations to its sandbox, such as via Powerbox or security-scoped bookmarks, until relaunched.

For detailed descriptions of the methods, constants, and entitlements to use for implementing security-scoped bookmarks in your app, read *NSURL Class Reference*, and read Enabling Security-Scoped Bookmark and URL Access in *Entitlement Key Reference*.

Note: The Core Foundation framework also provides equivalent C functions for working with security-scoped bookmarks. For details, see the documentation for `CFURLCreateBookmarkData`, `CFURLCreateByResolvingBookmarkData`,`CFURLStartAccessingSecurityScopedResource`, and `CFURLStopAccessingSecurityScopedResource` in *CFURL Reference*.

App Sandbox and Code Signing

After you enable App Sandbox and specify other entitlements for a target in your Xcode project, you must code sign the project. Take note of the distinction between how you set entitlements and how you set a code signing identity:

- Entitlements must be set on a target-by-target basis using the Xcode target editor.
- The code signing identity is typically set for a project as a whole using the Xcode project build settings, but may be overridden on a per-target basis, if desired.

You must perform code signing because entitlements (including the special entitlement that enables App Sandbox) are built into an app's code signature. From another perspective, an unsigned app is not sandboxed and has only default entitlements, regardless of settings you've applied in the Xcode target editor.

OS X enforces a tie between an app's container and the app's code signature. This important security feature ensures that no other sandboxed app can access your container. The mechanism works as follows:

1. When the system creates a container for an app, it sets an access control list (ACL) on that container. The initial access control entry in that list contains the app's Designated Requirement (DR), which is part of the app's signature that describes how future versions of the app can be recognized.
2. Each time an app with the same bundle ID launches, the system checks that the app's code signature matches the designated requirements specified in one of the entries in the container's ACL. If the system does not find a match, it prevents the app from launching.

OS X's enforcement of container integrity impacts your development and distribution cycle. This is because, in the course of creating and distributing an app, the app is code signed using various signatures. Here's how the process works:

1. Before you create a project, you obtain three code signing certificates from Apple: a development certificate, a distribution certificate, and (optionally) a Developer ID certificate. (To learn how to obtain these code signing certificates, read *App Distribution Guide*.)

When used in conjunction with the corresponding private keys from your keychain, these certificates form three separate digital identities. For development and testing, you sign your app with your development identity. When you submit a version to the app store, you use your distribution identity. If you are distributing a version outside the app store, you use your Developer ID identity.

2. When the Mac App Store distributes your app, it is signed with an Apple code signature.

For testing and debugging, you may want to run both versions of your app: the version you sign and the version Apple signs. But OS X sees the Apple-signed version of your app as an intruder and won't allow it to launch: Its code signature does not match the one expected by your app's existing container.

If you try to run the Apple-signed version of your app, you get a crash report containing a statement similar to this:

```
Exception Type:  EXC_BAD_INSTRUCTION (SIGILL)
```

The solution is to adjust the access control list (ACL) on your app's container to recognize the Apple-signed version of your app. Specifically, you add the designated code requirement of the Apple-signed version of your app to the app container's ACL.

To adjust an ACL to recognize an Apple-signed version of your app

1. Open Terminal (in `/Applications/Utilities`).
2. Open a Finder window that contains the Apple-signed version of your app.
3. In Terminal, enter the following command:

   ```
   asctl container acl add -file <path/to/app>
   ```

   In place of the `<path/to/app>` placeholder, substitute the path to the Apple-signed version of your app. Instead of manually typing the path, you can drag the app's Finder icon to the Terminal window.

The container's ACL now includes the designated code requirements for both versions of your app. OS X then allows you to run either version of your app.

You can use this same technique to share a container between (1) a version of an app that you initially signed with a development identity, such as the one you used in App Sandbox Quick Start (page 8), and (2) a released version downloaded from the Mac App Store.

You can view the list of code requirements in a container's ACL. For example, after adding the designated code requirement for the Apple-signed version of your app, you can confirm that the container's ACL lists two permissible code requirements.

To display the list of code requirements in a container's ACL

1. Open Terminal (in `/Applications/Utilities`).
2. In Terminal, enter the following command:

   ```
   asctl container acl list -bundle <container name>
   ```

   In place of the `<container name>` placeholder, substitute the name of your app's container directory. (The name of your app's container directory is typically the same as your app's bundle identifier.)

For more information about working with App Sandbox container access control lists and their code requirements, read the man page for the `asctl` (App Sandbox control) tool.

External Tools, XPC Services, and Privilege Separation

Some app operations are more likely to be targets of malicious exploitation. Examples are the parsing of data received over a network, and the decoding of video frames. By using XPC, you can improve the effectiveness of the damage containment offered by App Sandbox by separating such potentially dangerous activities into their own address spaces.

Your app can also launch existing helper apps using launch services, but only if certain conditions are met.

Important: If you are submitting your app to the Mac App Store, you must verify that any embedded helper tools or helper apps are also properly sandboxed.

To do this, run the following command on each embedded executable in your app bundle and confirm that each one has an App Sandbox entitlement:

```
codesign -dvvv --entitlements :- executable_path
```

where `executable_path` is the complete path to an executable binary in your app bundle.

The sections below explain these concepts in more detail.

XPC Services

XPC is an OS X interprocess communication technology that complements App Sandbox by enabling privilege separation. *Privilege separation*, in turn, is a development strategy in which you divide an app into pieces according to the system resource access that each piece needs. The component pieces that you create are called *XPC services* .

You create an XPC service as an individual target in your Xcode project. Each service gets its own sandbox—specifically, it gets its own container and its own set of entitlements.

Note: If you are distributing your app in the Mac App Store, XPC services must be sandboxed. For apps distributed elsewhere, sandboxing XPC services is strongly recommended.

In addition, an XPC service that you include with your app is accessible only by your app. These advantages add up to making XPC the best technology for implementing privilege separation in an OS X app.

By contrast, a child process created by using the `posix_spawn` function, by calling `fork` and `exec` (discouraged), or by using the `NSTask` class simply inherits the sandbox of the process that created it. You cannot configure a child process's entitlements. For these reasons, child processes do not provide effective privilege separation.

To use XPC with App Sandbox:

- Confer minimal privileges to each XPC service, according to its needs.
- Design the data transfers between the main app and each XPC service to be secure.
- Structure your app's bundle appropriately.

The life cycle of an XPC service, and its integration with Grand Central Dispatch (GCD), is managed entirely by the system. To obtain this support, you need only to structure your app's bundle correctly.

For more on XPC, see Creating XPC Services in *Daemons and Services Programming Guide* .

Launching Helpers with Launch Services

A sandboxed app is allowed to launch a helper using Launch Services if at least one of these conditions has been met:

- Both the app and helper pass the Gatekeeper assessment. By default that means both are signed by the Mac App Store or with a Developer ID.

  Note: This does *not* include your development ("Mac Developer") or distribution ("3rd Party Mac Developer Application") signing identities.

- The app is installed in `/Applications` and the app bundle and all contents are owned by root.
- The helper has been (manually) run at least once by the user.

If none of these conditions have been met, you'll see errors like the following:

- "Not allowing process 19920 to launch '/Applications/Main.app/Contents/Resources/Helper.app' because the security assessment verdict was denied."

  This message means that the Gatekeeper assessment was denied. You can confirm that with the `spctl` tool as follows:

  ```
  $ spctl --assess -vvvv /Applications/Main.app/


  /Applications/Main.app/: rejected

  origin=Mac Developer: Developer Name


  $ spctl --assess -vvvv
  /Applications/Main.app/Contents/Resources/Helper.app/


  /Applications/Main.app/Contents/Resources/Helper.app/: rejected

  origin=Mac Developer: Developer Name
  ```

- "The application “Helper” could not be launched because it is corrupt."

  "The operation couldn’t be completed. (OSStatus error -10827.)"

  This is the typical error if none of the above conditions have been fulfilled.

The results are the same whether you use Launch Services directly (by calling `LSOpenCFURLRef`, for example) or indirectly (by calling the `launchApplicationAtURL:options:configuration:error:` method in `NSWorkspace`, for example).

In addition, upon failure, in OS X v10.7.5 and earlier, you will also see a bogus `deny file-write-data /Applications/Main.app/Contents/Resources/Helper.app` sandbox violation. This error has no functional impact and can be ignored.

IPC and POSIX Semaphores and Shared Memory

Normally, sandboxed apps cannot use Mach IPC, POSIX semaphores and shared memory, or UNIX domain sockets (usefully). However, by specifying an entitlement that requests membership in an application group, an app can use these technologies to communicate with other members of that application group.

Note: System V semaphores are not supported in sandboxed apps.

UNIX domain sockets are straightforward; they work just like any other file.

Any semaphore or Mach port that you wish to access within a sandboxed app must be named according to a special convention:

- POSIX semaphores and shared memory names must begin with the application group identifier, followed by a slash (`/`), followed by a name of your choosing.
- Mach port names must begin with the application group identifier, followed by a period (`.`), followed by a name of your choosing.

For example, if your application group's name is `Z123456789.com.example.app-group`, you might create two semaphore named `Z123456789.myappgroup/rdyllwflg` and `Z123456789.myappgroup/bluwhtflg`. You might create a Mach port named `Z123456789.com.example.app-group.Port_of_Kobe`.

Note: The maximum length of a POSIX semaphore name is only 31 bytes, so if you need to use POSIX semaphores, you should keep your app group names *short*.

To learn more about application groups, read The Application Group Container Directory (page 18), then read Adding an Application to an Application Group in *Entitlement Key Reference*.

Designing for App Sandbox

There's a common, basic workflow for designing or converting an app for App Sandbox. The specific steps to take for your particular app, however, are as unique as your app. To create a work plan for adopting App Sandbox, use the process outlined here, along with the conceptual understanding you have from the earlier chapters in this document.

Six Steps for Adopting App Sandbox

The workflow to convert an OS X app to work in a sandbox typically consists of the following six steps:

1. Determine whether your app is suitable for sandboxing.
2. Design a development and distribution strategy.
3. Resolve API incompatibilities.
4. Apply the App Sandbox entitlements you need.
5. Add privilege separation using XPC.
6. Implement a migration strategy.

Note: It is not sufficient to perform this task for the main app in your app bundle. For apps distributed through the Mac App Store, all included helper apps and tools must also be sandboxed. For apps distributed through other mechanisms, you should sandbox each executable in your app bundle if at all possible.

For a list of all executable binaries in your app bundle, type the following command in Terminal:

```
find -H YourAppBundle.app -print0 | xargs -0 file | grep "Mach-O .*executable"
```

where `YourAppBundle.app` should be replaced by the path to your app bundle.

Determine Whether Your App Is Suitable for Sandboxing

Most OS X apps are fully compatible with App Sandbox. If you need behavior in your app that App Sandbox does not allow, consider an alternative approach. For example, if your app depends on hard-coded paths to locations in the user's home directory, consider the advantages of using Cocoa and Core Foundation path-finding APIs, which use the sandbox container instead.

If you choose to not sandbox your app now, or if you determine that you need a temporary exception entitlement, use Apple's bug reporting system to let Apple know what's not working for you. Apple considers feature requests as it develops the OS X platform. Also, if you request a temporary exception, be sure to use the Review Notes field in iTunes Connect to explain why the exception is needed.

The following app behaviors are *incompatible* with App Sandbox:

- Use of Authorization Services

  With App Sandbox, you cannot do work with the functions described in *Authorization Services C Reference* .

- Use of accessibility APIs in assistive apps

  With App Sandbox, you can and should enable your app for accessibility, as described in *Accessibility Programming Guide for OS X* . However, you cannot sandbox an assistive app such as a screen reader, and you cannot sandbox an app that controls another app.

- Sending Apple events to arbitrary apps

  With App Sandbox, you can receive Apple events and respond to Apple events, but you cannot send Apple events to arbitrary apps.

  However, for applications that specifically provide scripting access groups, you can send appropriate Apple events to those apps if your app includes a scripting targets entitlement.

  For other applications, by using a temporary exception entitlement, you can enable the sending of Apple events to a list of specific apps that you specify, as described in *Entitlement Key Reference* .

Finally, your app can use the subclasses of `NSUserScriptTask` class to run user-provided AppleScript scripts out of a special directory, `NSApplicationScriptsDirectory` (`~/Library/Application Scripts/`*code-signing-identifier*`/`). Although your app can read files within this directory, it cannot write files into this directory; the user must manually place scripts here. For details, see the documentation for `NSUserScriptTask` and *WWDC 2012: Secure Automation Techniques in OS X*.

- Sending user-info dictionaries in distributed notifications to other tasks

  With App Sandbox, you *cannot* include a `userInfo` dictionary when posting to an `NSDistributedNotificationCenter` object for messaging other tasks. (You *can*, as usual, include a `userInfo` dictionary when messaging other parts of your app by way of posting to an `NSNotificationCenter` object.)

- Loading kernel extensions

  Loading of kernel extensions is prohibited with App Sandbox.

- Simulation of user input in Open and Save dialogs

  If your app depends on programmatically manipulating Open or Save dialogs to simulate or alter user input, your app is unsuitable for sandboxing.

- Accessing or setting preferences on other apps

  With App Sandbox, each app maintains its preferences inside its container. Normally, your app has no access to the preferences of other apps.

  However, if your app requires access to the preferences files of other applications, there are temporary exception entitlements available that allow you to specify a list of named preference domains that your app needs to access. For details, see *Entitlement Key Reference*.

- Configuring network settings

  With App Sandbox, your app cannot modify the system's network configuration (whether with the System Configuration framework, the CoreWLAN framework, or other similar APIs) because doing so requires administrator privileges.

- Terminating other apps

  With App Sandbox, you cannot use the `NSRunningApplication` class to terminate other apps.

Resolve API Incompatibilities

If you are using OS X APIs in ways that were not intended, or in ways that expose user data to attack, you may encounter incompatibilities with App Sandbox. This section provides some examples of app design that are incompatible with App Sandbox and suggests what you can do instead.

Opening, Saving, and Tracking Documents

If you are managing documents using any technology other than the `NSDocument` class, you should convert to using this class to benefit from its built-in App Sandbox support. The `NSDocument` class automatically works with Powerbox. `NSDocument` also provides support for keeping documents within your sandbox if the user moves them using the Finder.

Remember that the inheritance path of the `NSOpenPanel` and `NSSavePanel` classes is different when your app is sandboxed. See Open and Save Dialog Behavior with App Sandbox (page 22).

If you don't use the `NSDocument` class to manage your app's documents, you can craft your own file-system support for App Sandbox by using the `NSFileCoordinator` class and the `NSFilePresenter` protocol, but this requires a lot of extra work.

Retaining Access to File System Resources

If your app depends on persistent access to file system resources outside of your app's container, you need to adopt security-scoped bookmarks as described in Security-Scoped Bookmarks and Persistent Resource Access (page 23).

Creating a Login Item for Your App

To create a login item for your sandboxed app, use the `SMLoginItemSetEnabled` function (declared in `ServiceManagement/SMLoginItem.h`) as described in Adding Login Items Using the Service Management Framework.

(With App Sandbox, you cannot create a login item using functions in the `LSSharedFileList.h` header file. For example, you cannot use the function `LSSharedFileListInsertItemURL`. Nor can you manipulate the state of launch services, such as by using the function `LSRegisterURL`.)

Accessing User Data

Most OS X path-finding APIs return paths relative to the container instead of relative to the user's home directory. If your app, before you sandbox it, accesses locations in the user's actual home directory (~) and you are using Cocoa or Core Foundation APIs, then, after you enable sandboxing, your path-finding code automatically uses your app's container instead.

For first launch of your sandboxed app, OS X automatically migrates your app's main preferences file. If your app uses additional support files, perform a one-time migration of those files to the container, as described in Migrating an App to a Sandbox (page 39).

If you are using a POSIX function such as `getpwuid` to obtain the path to the user's actual home directory from directory services (rather than by using the `HOME` environment variable), consider instead using a Cocoa or Core Foundation symbol such as the `NSHomeDirectory` function. By using Cocoa or Core Foundation, you support the App Sandbox restriction against directly accessing the user's home directory.

If your app requires access to the user's home directory in order to function, let Apple know about your needs using the Apple bug reporting system. In addition, be sure to follow the guidance regarding entitlements provided on the iTunes Connect website.

Accessing Preferences of Other Apps

Because App Sandbox directs path-finding APIs to the container for your app, reading or writing to the user's preferences takes place within the container. Preferences for other sandboxed apps are inaccessible. Preferences for apps that are not sandboxed are placed in the `~/Library/Preferences` directory, which is also inaccessible to your sandboxed app.

If your app requires access to another app's preferences in order to function—for example, if it requires access to the playlists that a user has defined for iTunes—let Apple know about your needs using the Apple bug reporting system. In addition, be sure to follow the guidance regarding entitlements provided on the iTunes Connect website.

Using HTML5 Embedded Video in Web Views

If you are compiling an app that uses the WebKit framework, and your target is OS X v10.7, you must also link your app against the AV Foundation framework. If you do not do so, because of the way App Sandbox interacts with CoreMedia, your app will be unable to play HTML5 embedded videos.

This additional linking step is not required for apps that run only on OS X v10.8 and later.

Apply the App Sandbox Entitlements You Need

To adopt App Sandbox for a target in an Xcode project, apply the `<true/>` value to the `com.apple.security.app-sandbox entitlement` key for that target. Do this in the Xcode target editor by selecting the Enable App Sandboxing checkbox.

Apply other entitlements as needed. For a complete list, refer to *Entitlement Key Reference* .

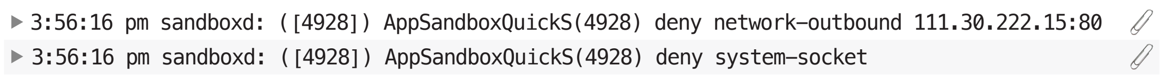
Important: App Sandbox protects user data most effectively when you minimize the entitlements you request. Take care not to request entitlements for privileges your app does not need. Consider whether making a change in your app could eliminate the need for an entitlement.

Here's a basic workflow to use to determine which entitlements you need:

1. Run your app and exercise its features.
2. In the Console app (available in `/Applications/Utilities/`), look for `sandboxd` violations in the All Messages system log query.

   Each such violation indicates that your app attempted to do something not allowed by your sandbox.

   Here's what a `sandboxd` violation looks like in Console:

   ```
   ▶ 3:56:16 pm sandboxd: ([4928]) AppSandboxQuickS(4928) deny network-outbound 111.30.222.15:80
   ▶ 3:56:16 pm sandboxd: ([4928]) AppSandboxQuickS(4928) deny system-socket
   ```

   Click the paperclip icon to the right of a violation message to view the backtrace that shows what led to the violation.
3. For each `sandboxd` violation you find, determine how to resolve the problem. In same cases, a simple change to your app, such as using your Container instead of other file system locations, solves the problem. In other cases, applying an App Sandbox entitlement using the Xcode target editor is the best choice.
4. Using the Xcode target editor, enable the entitlement that you think will resolve the violation.
5. Run the app and exercise its features again.

   Either confirm that you have resolved the `sandboxd` violation, or investigate further.

If you choose not to sandbox your app now or to use a temporary exception entitlement, use Apple's bug reporting system to let Apple know about the issue you are encountering. Apple considers feature requests as it develops the OS X platform. Also, be sure use the Review Notes field in iTunes Connect to explain why the exception is needed.

Add Privilege Separation Using XPC

When developing for App Sandbox, look at your app's behaviors in terms of privileges and access. Consider the potential benefits to security and robustness of separating high-risk operations into their own XPC services.

When you determine that a feature should be placed into an XPC service, do so by referring to Creating XPC Services in *Daemons and Services Programming Guide* .

Implement a Migration Strategy

Ensure that customers who are currently using a pre-sandbox version of your app experience a painless upgrade when they install the sandboxed version. For details on how to implement a container migration manifest, read Migrating an App to a Sandbox (page 39).

Migrating an App to a Sandbox

An app that is not sandboxed places its support files in locations that are inaccessible to a sandboxed version of the same app. For example, the typical locations for support files are shown here:

Path	Description
`~/Library/Application Support/<app_name>/`	Legacy location
`~/Library/Containers/<bundle_id>/Data/Library/Application Support/<app_name>/`	Sandbox location

As you can see, the sandbox location for the `Application Support` directory is within an app's container—thus allowing the sandboxed app unrestricted read/write access to those files. If you previously distributed your app without sandboxing and you now want to provide a sandboxed version, you must tell OS X to move your support files into their new, sandbox-accessible locations.

Note: The system automatically migrates your app's preferences file (`~/Library/Preferences/com.yourCompany.YourApp.plist`) on first launch of your sandboxed app. You do not need to explicitly request that it be migrated.

When a user launches any app, before the app starts running, OS X checks to see if the app is sandboxed. If so, OS X checks to see if a sandbox container directory exists for that particular app within the user's `Library/Containers` folder. If the app's container directory already exists, then no migration is needed, and your app begins running immediately.

However, if the app's container directory does not exist, OS X creates the missing sandbox container directory and then migrates the per-user preference file for your app, along with any files that you have specifically listed in the app's *container migration manifest* .

A *container migration manifest* is a special property list file containing an array of strings that identify the support files and directories you want to migrate when a user first launches the sandboxed version of your app. The file must be named `container-migration.plist`, and must appear in the `Contents/Resources` directory within your app bundle.

For each file or directory you specify for migration, you have a choice of allowing the system to place the item appropriately in your container or explicitly specifying the destination location.

OS X moves—it does not copy—the files and directories you specify in a container migration manifest. That is, the files and directories migrated into your app's container no longer exist at their original locations. In addition, container migration is a one-way process: You are responsible for providing a way to undo it, should you need to do so during development or testing. The section Undoing a Migration for Testing (page 42) provides a suggestion about this.

Creating a Container Migration Manifest

To support migration of app support files when a user first launches the sandboxed version of your app, create a container migration manifest.

To create and add a container migration manifest to an Xcode project

1. Add a property list file to the Xcode project.

   The Property List template is in the OS X "Resource" group in the file template dialog.

   Important: Be sure to name the file `container-migration.plist` spelled and lowercased exactly this way.

2. Add a `Move` property to the container migration manifest.

   The `Move` property contains an array of files to move into your container directory. This property is usually the lone top-level key in a container migration manifest. You add it to the empty file as follows:

   - Right-click the empty editor for the new `.plist` file, then choose Add Row.
   - In the Key column, enter `Move` as the name of the key.

     You must use this exact casing and spelling.
   - In the Type column, choose `Array`.

3. Optionally add a `Symlink` property to the container migration manifest.

   The `Symlink` property contains an array of files to link into your container directory. The resulting symbolic links in your container directory point to files elsewhere.

   Note: If you want these symbolic links to be useful, you must combine them with temporary exception entitlements to grant access to whatever files or folders the symbolic links point to.

4. Add a string to the `Move` (or `Symlink`) array for the first file or folder you want to migrate.

   For example, suppose you want to migrate your `Application Support` directory (along with its contained files and subdirectories) to your container. If your directory is called `App Sandbox Quick Start` and is currently within the `~/Library/Application Support` directory, use the following string as the value for the new property list item:

   ```
   ${ApplicationSupport}/App Sandbox Quick Start
   ```

   No trailing slash character is required, and space characters are permitted. The search-path constant in the path is equivalent to `~/Library/Application Support`. This constant is described, along with other commonly used directories, in Use Variables to Specify Support-File Directories (page 44).

Similarly, add additional strings to identify the original (before sandboxing) paths of additional files or folders you want to migrate.

When you specify a directory to be moved, keep in mind that the move is recursive—it includes all the subdirectories and files within the directory you specify.

Before you first test a migration manifest, provide a way to undo the migration, such as suggested in Undoing a Migration for Testing (page 42).

To test a container migration manifest

1. In the Finder, open two windows as follows:
   - In one window, view the contents of the `~/Library/Containers/` directory.
   - In the other window, view the contents of the directory containing the support files named in the container migration manifest—that is, the files you want to migrate.

2. Build and run the Xcode project.

Upon successful migration, the support files disappear from the original (nonsandbox) directory and appear in your app's container.

If you want to alter the arrangement of support files during migration, use a slightly more complicated `.plist` structure. Specifically, for a file or directory whose migration destination you want to control, provide both a starting and an ending path. The ending path is relative to the `Data` directory in your container. In specifying an ending path, you can use any of the search-path constants described in Use Variables to Specify Support-File Directories (page 44).

If your destination path specifies a custom directory (one that isn't part of a standard container), the system creates the directory during migration.

The following task assumes that you're using the Xcode property list editor and working with the container migration manifest you created earlier in this chapter.

To control the destination of a migrated file or directory

1. In the container migration manifest, add a new item to the `Move` (or `Symlink`) array.
2. In the Type column, choose Array.
3. Add two strings as children of the new array item.
4. In the top string of the pair, specify the origin path of the file or directory you want to migrate.
5. In the bottom string of the pair, specify the destination (sandbox) custom path for the file or directory you want to migrate.

File migration proceeds from top-to-bottom through the container migration manifest. Take care to list items in an order that works. For example, suppose you want to move your entire `Application Support` directory as-is, except for one file. You want that file to go into a new directory parallel to `Application Support` in the container.

For this approach to work, you must specify the individual file move before you specify the move of the `Application Support` directory—that is, specify the individual file move higher in the container migration manifest. (If `Application Support` were specified to be moved first, the individual file would no longer be at its original location at the time the migration process attempted to move it to its new, custom location in the container.)

Undoing a Migration for Testing

When testing migration of support files, you may find it necessary to perform migration more than once. To support this, you need a way to restore your starting directory structures—that is, the structures as they exist prior to migration.

One way to do this is to make a copy of the directories to migrate, before you perform a first migration. Save this copy in a location unaffected by the migration manifest. The following task assumes you have created this sort of backup copy.

To manually undo a container migration for testing purposes

1. Manually copy the files and directories—those specified in the manifest—from your backup copy to their original (premigration) locations. (Be careful not to remove the files from your backup copy as you do so.)
2. Delete your app's container.

The next time you launch the app, the system recreates the container and migrates the support files according to the current version of the container migration manifest.

An Example Container Migration Manifest

Listing 4-1 shows an example manifest as viewed in a text editor.

Listing 4-1 An example container migration manifest

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>Move</key>
   <array>
      <string>${Library}/MyApp/MyConfiguration.plist</string>
      <array>
         <string>${Library}/MyApp/MyDataStore.xml</string>
         <string>${ApplicationSupport}/MyApp/MyDataStore.xml</string>
     </array>
  </array>
</dict>
</plist>
```

This manifest specifies the migration of two items from the user's `Library` directory to the app's container. For the first item, `MyConfiguration.plist`, only the origin path is specified, leaving it to the migration process to place the file appropriately.

For the second item, `MyDataStore.xml`, both an origin and a custom destination path are specified.

The `${Library}` and `${ApplicationSupport}` portions of the paths are variables you can use as a convenience. For a list of variables you can use in a container migration manifest, see Use Variables to Specify Support-File Directories (page 44).

Use Variables to Specify Support-File Directories

When you specify a path in a container migration manifest, you can use certain variables that correspond to commonly used support file directories. These variables work in origin and destination paths, but the path that a variable resolves to depends on the context. Refer to Table 4-1.

Table 4-1 How system directory variables resolve depending on context

Context	Variable resolves to
Origin path	Home-relative path (relative to the ~ directory)
Destination path	Container-relative path (relative to the `Data` directory in the container)

The variables you can use for specifying support-file directories are described in Table 4-2 (page 44). For an example of how to use these variables, see Listing 4-1 (page 43).

You can also use a special variable that resolves to your app's bundle identifier, allowing you to conveniently incorporate it into an origin or destination path. This variable is `${BundleId}`.

Table 4-2 Variables for support-file directories

Variable	Directory
`${Application-Support}`	The directory containing application support files. Corresponds to the `NSApplicationSupportDirectory` search-path constant.
`${Autosaved-Information}`	The directory containing the user's autosaved documents. Corresponds to the `NSAutosavedInformationDirectory` search-path constant.
`${Caches}`	The directory containing discardable cache files. Corresponds to the `NSCachesDirectory` search-path constant.
`${Document}` `${Documents}`	Each variable corresponds to the directory containing the user's documents. Corresponds to the `NSDocumentDirectory` search-path constant.

Variable	Directory
`${Home}`	The current user's home directory. Corresponds to the directory returned by the `NSHomeDirectory` function. When in a destination path in a manifest, resolves to the container directory.
`${Library}`	The directory containing application-related support and configuration files. Corresponds to the `NSLibraryDirectory` search-path constant.

Document Revision History

This table describes the changes to *App Sandbox Design Guide* .

Date	Notes
2014-02-11	Corrected the explanation of the stopAccessingSecurityScopedResource method.
2013-10-22	Added information about app group container behavior in OS X v10.9.
2013-08-08	Removed inaccurate guidance about handling issues where an app needs access to another app's preferences.
2013-03-14	Added information about related items in OS X v10.8.
2012-09-19	Clarified information about launching external tools.
2012-07-23	Added an explanation of app group containers.
2012-05-14	Improved the explanation of security-scoped bookmarks in Security-Scoped Bookmarks and Persistent Resource Access (page 23); updated that section for OS X v10.7.4. Added a brief section in the Designing for App Sandbox chapter: Retaining Access to File System Resources (page 35). Improved the discussion in Opening, Saving, and Tracking Documents (page 35), adding information about using file coordinators. Corrected the information in Creating a Login Item for Your App (page 35).

Date	Notes
2012-03-14	Improved explanation of security-scoped bookmarks in Security-Scoped Bookmarks and Persistent Resource Access (page 23). Clarified the explanation of the container directory in The App Sandbox Container Directory (page 16)
2012-02-16	Updated for OS X v10.7.3, including an explanation of how to use security-scoped bookmarks. Added a section explaining how to provide persistent access to file-system resources, Security-Scoped Bookmarks and Persistent Resource Access (page 23). Expanded the discussion in Powerbox and File System Access Outside of Your Container (page 19) to better explain how user actions expand your app's file system access. Added a section detailing the changes in behavior of Open and Save dialogs, Open and Save Dialog Behavior with App Sandbox (page 22).
2011-09-27	New document that explains Apple's security technology for damage containment, and how to use it. Portions of this document were previously published in *Code Signing and Application Sandboxing Guide*.