On File System Permissions

This thread has been locked by a moderator.

2.4k

Modern versions of macOS use a file system permission model that's *far more* complex than the traditional BSD `rwx` model, and this post is my attempt at explaining that new model. If you have a question about this, post it here on DevForums, tagging your thread with *Files and Storage* so that I see it.

Share and Enjoy

—

Quinn "The Eskimo!" @ Developer Technical Support @ Apple

```
let myEmail = "eskimo" + "1" + "@" + "apple.com"
```

On File System Permissions

Modern versions of macOS have four different file system permission mechanisms:

- Traditional BSD permissions
- Access control lists (ACLs)
- App Sandbox
- Mandatory access control (MAC)

The first two were introduced a long time ago and rarely trip folks up. The second two are newer, more complex, and specific to macOS, and thus are the source of some confusion. This post is my attempt to clear that up.

Error Codes

App Sandbox and the mandatory access control system are both implemented using macOS's sandboxing feature. When a file system operation fails, check the error to see whether it was blocked by this sandboxing feature. If an operation was blocked by BSD permissions or ACLs, it fails with `EACCES` (*Permission denied*, 13). If it was blocked by something else, it'll fail with `EPERM` (*Operation not permitted*, 1).

If you're using Foundation's `FileManager`, these error are reported as Foundation errors, for example, the `NSFileReadNoPermissionError` error. To recover the underlying error, get the `NSUnderlyingErrorKey` property from the info dictionary.

App Sandbox

File system access within the App Sandbox is controlled by two factors. The first is the entitlements on the main executable. There are three relevant groups of entitlements:

- The `com.apple.security.app-sandbox` entitlement enables the App Sandbox. This denies access to all file system locations except those on a built-in allowlist (things like `/System`) or within the app's containers.
- The various "standard location" entitlements extend the sandbox to include their corresponding locations.
- The various "file access temporary exceptions" entitlements extend the sandbox to include the items listed in the entitlement.

The second factor is dynamic sandbox extensions. The system issues these extensions to your sandbox based on user behaviour. For example, if the user selects a file in the open panel, the system issues a sandbox extension to your process so that it can access that file. The type of extension is determined by the main executable's entitlements:

- `com.apple.security.files.user-selected.read-only` results in an extension that grants read-only access.
- `com.apple.security.files.user-selected.read-write` results in an extension that grants read/write access.

These dynamic sandbox extensions are tied to your process; they go away when your process terminates. To maintain persistent access to an item, use a security-scoped bookmark. See Security-Scoped Bookmarks and Persistent Resource Access.

If you have access to a directory — regardless of whether that's via an entitlement or a dynamic sandbox extension — then, in general, you have access to all items in the hierarchy rooted at that directory. This does not overrule the MAC protection discussed below. For example, if the user grants you access to `~/Library`, that does not give you access to `~/Library/Mail` because the latter is protected by MAC.

Finally, the discussion above is focused on a new sandbox, the thing you get when you launch a sandboxed app from the Finder. If a sandboxed process starts a child process, that child process inherits its sandbox from its parent. For information on what happens in that case, see the Note box in Enabling App Sandbox Inheritance.

IMPORTANT The child process inherits its parent process's sandbox regardless of whether it has the `com.apple.security.inherit` entitlement. That entitlement exists primarily to act as a marker for App Review. App Review requires that all main executables have the `com.apple.security.app-sandbox` entitlement, and that entitlements starts a new sandbox by default. Thus, any helper tool inside your app needs the `com.apple.security.inherit` entitlement to trigger inheritance. However, if you're not shipping on the Mac App Store you can leave off both of these entitlement and the helper process will inherit its parent's sandbox just fine. The same applies if you run a built-in executable, like `/bin/sh`, as a child process.

When the App Sandbox blocks something, it typically generates a sandbox violation report. For information on how to view these reports, see Viewing Sandbox Violation Reports.

To learn more about the App Sandbox, see the App Sandbox Design Guide and related documents (most notably the Entitlement Key Reference). For information about how to embed a helper tool in a sandboxed app, see Embedding a Command-Line Tool in a Sandboxed App.

Mandatory Access Control

Mandatory access control (MAC) has been a feature of macOS for *many* releases, but it's become a *lot* more prominent since macOS 10.14. There are many flavours of MAC but the ones you're most likely to encounter are:

- Full Disk Access (since 10.14)
- Files and Folders (since 10.15)
- Data Vaults (see below)

Mandatory access control, as the name suggests, is mandatory; it's not an opt-in like the App Sandbox. Rather, all processes on the system, including those running as root, as subject to MAC.

Data Vaults are not a third-party developer opportunity. See this post if you're curious.

In the Full Disk Access and Files and Folders case users grant a program a MAC privilege using System Preferences > Security & Privacy > Privacy. Some MAC privileges are per user (Files and Folders) and some are system wide (Full Disk Access). If you're not sure, run this simple test:

1. On a Mac with two users, log in as user A and enable the MAC privilege for a program.
2. Now log in as user B. Does the program have the privilege?

If a process tries to access an item restricted by MAC, the system may prompt the user to grant it access there and then. For example, if an app tries to access the desktop, you'll see an alert like this:

```
“AAA” would like to access files in your Desktop folder.

[Don’t Allow] [OK]
```

To customise this message, set properties in your `Info.plist`. See the *Files and Folders* topic on this page.

This system only displays this alert once. It remembers the user's initial choice and returns the same result thereafter. This relies on your code having a stable code signing identity. If your code is unsigned, or signed ad hoc ("Signed to run locally" in Xcode parlance), the system can't tell that version N+1 of your code is the same as version N, and thus you'll encounter excessive prompts.

Note For information about how that works, see TN3127 Inside Code Signing: Requirements.

The Files and Folders prompts only show up if the process is running in a GUI login session. If not, the operation is allowed or denied based on existing information. If there's no existing information, the operation is denied by default.

On managed systems the site admin can use the `com.apple.TCC.configuration-profile-policy` payload to assign MAC privileges.

For testing purposes you can reset parts of TCC using the `tccutil` command-line tool. For general information about that tool, see its man page. For a list of TCC service names, see the posts on this thread.

Note *TCC* stands for *transparency, consent, and control*. It's the subsystem within macOS that manages the privileges visible in System Preferences > Security & Privacy > Privacy. TCC has no API surface, but you see its name in various places, including the above-mentioned configuration profile payload and command-line tool, and the name of its accompanying daemon, `tccd`.

While `tccutil` is an easy way to do basic TCC testing, the most reliable way to test TCC is in a VM, restoring to a fresh snapshot between each test. If you want to try this out, crib ideas from Testing a Notarised Product.

The MAC privilege mechanism is heavily dependent on the concept of *responsible code*. For example, if an app contains a helper tool and the helper tool triggers a MAC prompt, we want:

- The app's name and usage description to appear in the alert.
- The user's decision to be recorded for the whole app, not that specific helper tool.
- That decision to show up in System Preferences under the app's name.

For this to work the system must be able to tell that the app is the responsible code for the helper tool. The system has various heuristics to determine this and it works reasonably well in most cases. However, it's possible to break this link. I haven't fully research this but my experience is that this most often breaks when the child process does something 'odd' to break the link, such as trying to daemonise itself.

Scripting

MAC presents some serious challenges for scripting because scripts are run by interpreters and the system can't distinguish file system operations done by the interpreter from those done by the script. For example, if you have a script that needs to manipulate files on your desktop, you wouldn't want to give the interpreter that privilege because then *any* script could do that.

The easiest solution to this problem is to package your script as a standalone program that MAC can use for its tracking. This may be easy or hard depending on the specific scripting environment. For example, AppleScript makes it easy to export a script as a signed app, but that's not true for shell scripts.

TCC and Main Executables

TCC expects its bundled clients — apps, app extensions, and so on — to use a native main executable. That is, it expects the `CFBundleExecutable` property to be the name of a Mach-O executable. If your product uses a script as its main executable, you are likely to encounter TCC problems. To resolve these, switch to using a Mach-O executable.

Revision History

- **2021-04-26** Added an explanation of the *TCC* initialism. Added a link to Viewing Sandbox Violation Reports. Added the *TCC and Main Executables* section. Made significant editorial changes.
- **2022-01-10** Added a discussion of the file system hierarchy.
- **2021-04-26** First posted.

Files and Storage Privacy

Reply

Posted 1 year ago by eskimo

Add a Comment