# Resolving Gatekeeper Problems Caused by Dangling Load Command Paths

This thread has been locked by a moderator.

377

This post is part of a cluster of posts related to the trusted execution system. If you found your way here directly, I recommend that you start at the top.

Share and Enjoy

—

Quinn “The Eskimo!” @ Developer Technical Support @ Apple

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

## Resolving Gatekeeper Problems Caused by Dangling Load Command Paths

Gatekeeper strives to ensure that only trusted software runs on a user’s Mac. It’s important that your code pass Gatekeeper. If not, you’re likely to lose a lot of customers, and your users’ hard-won trust.

The most common reason for Gatekeeper to block an app is a dangling load command. To understand this issue, you first need to understand library validation.

Library validation is an important security feature on macOS. If library validation is enabled on an executable, the process running that executable can only load code signed by Apple or with the same Team ID as the executable. This prevents a wide range of dynamic library impersonation attacks.

Library validation is enabled by the Hardened Runtime but you may opt out of it using the Disable Library Validation Entitlement (`com.apple.security.cs.disable-library-validation`) entitlement.

**IMPORTANT** Leave library validation enabled. Only disable it if your app needs to load plug-ins from other third-party developers.

When Gatekeeper checks an app it looks at the state of library validation. If library validation is disabled, Gatekeeper does an extensive check of the app’s Mach-O images in an attempt to prevent dynamic library impersonation attacks. If library validation is enabled, Gatekeeper skips that check because library validation prevents such attacks. The upshot of this is that, if you disable library validation, it’s harder to pass Gatekeeper.

**Note** The Swift Package Manager creates a dangling load command when you build a command-line tool for the Mac. For more on that issue, see the Compiler adds RPATH to executable, making macOS Gatekeeper angry thread on Swift Forums.

### Find Dangling Load Command Paths

If your app is rejected by Gatekeeper, look in the system log for entries like this:

```
type: error
time: 2022-05-11 15:00:36.296159 -0700
process: XprotectService
subsystem: com.apple.xprotect
category: xprotect
message: File /Applications/DanglingRPath.app/Contents/MacOS/DanglingRPath failed on rPathCmd
/Users/mrgumby/Work/TrustedExecutionFailures/CoreWaffleVarnishing.framework/Versions/A/CoreWaffleVarnishing (rpath resolved to:
(path not found), bundleURL: /Applications/DanglingRPath.app)
```

In this example the entry mentions `rPathCmd`, and that’s a good way to search for such problems. Also search for `loadCmd`, which indicates a similar problem.

**IMPORTANT** If the paths in the log entry are all `<private>`, enable private data in the system log. For information on how to do that, see Recording Private Data in the System Log. For general information about the system log, see Your Friend the System Log.

In this example Gatekeeper has rejected the DanglingRPath app because:

- It references a framework, CoreWaffleVarnishing, using an rpath-relative reference.
- The rpath includes an entry that points outside of the app’s bundle, to a directory called `/Users/mrgumby/Work`.

This opens the app up to a dynamic library impersonation attack. If an attacker placed a malicious copy of `CoreWaffleVarnishing` in `/Users/mrgumby/Work`, the DanglingRPath app would load it.

To find the offending rpath entry, run `otool` against each Mach-O image in the app looking for a dangling `LC_RPATH` load command. For example:

```
% otool -l DanglingRPath.app/Contents/MacOS/DanglingRPath | grep -B 1 -A 2 LC_RPATH
Load command 18
          cmd LC_RPATH
      cmdsize 48
         path @executable_path/../Frameworks (offset 12)
Load command 19
          cmd LC_RPATH
      cmdsize 56
         path /Users/mrgumby/Work (offset 12)
```

This app has two `LC_RPATH` commands. The one, for `@executable_path/../Frameworks`, is fine: It points to a location within the app’s bundle. In contrast, the one for `/Users/mrgumby/Work` is clearly dangling.

In this example, the dangling rpath entry is in the main executable but that’s not always the case; you might find it lurking in some deeply nested dynamic library.

Keep in mind that this is only an issue because the app has library validation disabled:

```
% codesign -d --entitlements - DanglingRPath.app
Executable=/Users/mrgumby/Desktop/DanglingRPath.app/Contents/MacOS/DanglingRPath
[Dict]
    [Key] com.apple.security.cs.disable-library-validation
    [Value]
        [Bool] true
```

If library validation were enabled, Gatekeeper would skip this check entirely, and thus the app would sail past Gatekeeper.

**IMPORTANT** In this example the app’s main executable has library validation disabled, but that’s not always the case. It’s common to see this problem in app’s that have multiple executables — the app itself and, say, one or two embedded helper tools — where one of the other executables has library validation disabled.

Finally, the above example is based on rpath-relative paths. If you see a log entry containing the text `loadCmd`, search for dangling paths in `LC_LOAD_DYLIB` load commands.

### Fix Dangling Load Command Paths

The best way to fix this problem is to not disable library validation. Library validation is an important security feature. By disabling it, you introduce this problem *and* reduce the security of your app. Conversely, by re-enabling it, you fix this problem and you improve security overall.

The only situation where it makes sense to disable library validation is if your app loads plug-ins from other third-party developers. In that case, fixing this problem requires you to find and eliminate all dangling load command paths. There are two possibilities here:

- The dangling load command path is in a Mach-O image that you built from source.
- Or it’s in a Mach-O image that you got from a vendor, for example, a library in some third-party SDK.

In the first case, solve this problem by adjusting your build system to not include the dangling load command path.

In the second case, work with the vendor to eliminate the dangling load command path.

If the vendor is unwilling to do this, it’s possible, as a last resort, to fix this problem by modifying the load commands yourself. For an example of how you might go about doing this, see Embedding Nonstandard Code Structures in a Bundle. And once you’re done, and your product has shipped, think carefully about whether you want to continue working with this vendor.

### Revision History

- **2022-06-13** Added a link to a related Swift Forums thread.
- **2022-05-20** First posted.

Gatekeeper Code Signing Notarization

Reply

Posted 4 months ago by eskimo

Add a Comment