Resolving Library Loading Problems

This thread has been locked by a moderator.

1.6k

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 Library Loading Problems

On macOS the dynamic linker is responsible for loading the dynamic libraries used by your process. There are two parts to this:

- When a process runs an executable, the dynamic linker loads all the libraries used by that executable, the libraries used by those libraries, and so on.
- A process may load libraries at runtime using APIs like `dlopen` and `NSBundle`. For information the `dlopen` and friends, see the `dlopen` man page.

The dynamic linker works closely with the trusted execution system to ensure that it only loads appropriate libraries. A critical concern is dynamic library impersonation attacks. If your program references a library, you want the dynamic linker to load that copy of the library, and not some other copy installed by an attacker.

The primary protection is library validation. If library validation is enabled on an executable, the trusted execution system only allows the process to load code signed by Apple or with the same Team ID as the executable.

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. Disabling library validation makes it harder to pass Gatekeeper. See Resolving Gatekeeper Problems Caused by Dangling Load Command Paths for the details.

When the dynamic linker fails to load a library it includes an explanation in the crash report. For example:

```
Termination Reason:    Namespace DYLD, Code 1 Library missing
Library not loaded: @rpath/libEtranger.dylib
Referenced from: /Users/DSER/vL/LinkEtranger.app/Contents/MacOS/LinkToEtranger
Reason: …
(terminated at launch; ignore backtrace)

Application Specific Information:
Library not loaded: @rpath/libEtranger.dylib
Referenced from: …
Reason: …
```

This explanation is likely to be truncated by the crash reporting system. To see the full log, run the app from Terminal:

```
% ./LinkToEtranger.app/Contents/MacOS/LinkToEtranger
dyld[79658]: Library not loaded: @rpath/libEtranger.dylib
  Referenced from: …/LinkToEtranger.app/Contents/MacOS/LinkToEtranger
  Reason: tried: '…/LinkToEtranger.app/Contents/MacOS/../frameworks/libEtr
    anger.dylib' (code signature in <E1ED3D14-CC5A-33BC-9B06-554A3BC12C51>
    '…/LinkToEtranger.app/Contents/frameworks/libEtranger.dylib' not valid
    for use in process: mapping process and mapped file (non-platform) have
    different Team IDs), '…/LinkToEtranger.app/Contents/MacOS/../
    frameworks/libEtranger.dylib' (code signature in <E1ED3D14-CC5A-33BC-
    9B06-554A3BC12C51> '…/LinkToEtranger.app/Contents/frameworks/
    libEtranger.dylib' not valid for use in process: mapping process and
    mapped file (non-platform) have different Team IDs), '/usr/local/lib/
    libEtranger.dylib' (no such file), '/usr/lib/libEtranger.dylib' (no
    such file)
zsh: abort      ./LinkToEtranger.app/Contents/MacOS/LinkToEtranger
```

The `Reason` line is super long, so break it up by attempt:

```
'…/LinkToEtranger.app/Contents/MacOS/../frameworks/libEtranger.dylib'
    (…),
'/usr/local/lib/libEtranger.dylib' (no such file),
'/usr/lib/libEtranger.dylib' (no such file)
```

Each entry starts with a place that the dynamic linker attempted to find library and then has text inside parentheses, like `no such file`, explaining what went wrong.

Note The exact format of these messages varies from release-to-release of macOS.

Many of these reasons are unrelated to the trusted execution system. For example, `no such file` means that the library isn't present on disk. There are, however, three common trusted execution issues:

- Library validation
- Use of an old macOS SDK
- Restricted entitlements on library code

For more information about the dynamic linker, see the `dyld` man page. Specifically, the `DYLD_PRINT_SEARCHING` environment variable is super useful when debugging library loading problems.

Library Validation

In any real world situation the `Reason` output from the dynamic linker is super long. To understand it better, break it up by attempt:

```
'…/LinkToEtranger.app/Contents/MacOS/../frameworks/libEtranger.dylib'
    (code signature in <E1ED3D14-CC5A-33BC-9B06-554A3BC12C51>
    '…/LinkToEtranger.app/Contents/frameworks/libEtranger.dylib'
    not valid for use in process: mapping process and mapped file
    (non-platform) have different Team IDs),
'/usr/local/lib/libEtranger.dylib' (no such file),
'/usr/lib/libEtranger.dylib' (no such file)
```

The dynamic linker looked in three different places:

- The app's `Frameworks` directory
- `/usr/local/lib`
- `/usr/lib`

The first one is the important one because its path matches the expected location of the library. And the dynamic linker has logged an excellent explanation of the problem:

```
code signature in … '…/LinkToEtranger.app/Contents/Frameworks/
libEtranger.dylib' not valid for use in process: mapping process
and mapped file (non-platform) have different Team IDs
```

In summary, the dynamic linker didn't load this copy of `libEtranger.dylib` because it's not a system library (`non-platform`) and it has a different Team ID from the process's main executable. A quick trip to `codesign` confirms this:

```
% codesign -d -vvv LinkToEtranger.app
…
TeamIdentifier=S4PME9E2Y8
…
% codesign -d -vvv LinkToEtranger.app/Contents/Frameworks/libEtranger.dylib
…
TeamIdentifier=VL95QP7560
…
```

How you fix this depends on the nature of your product. If this library is installed as part of your product, re-sign the library with a signing identity associated with your Team ID. Do this even if you didn't build the code yourself. After all, you were responsible for putting the library on the user's machine, and its signature should reflect that.

One other possibility is that you're building a program that supports plug-ins and thus you need to load a plug-in that was signed by another third-party developer. In this case the fix is to disable library validation by signing your executable with the Disable Library Validation Entitlement entitlement (`com.apple.security.cs.disable-library-validation`).

IMPORTANT Disabling library validation makes it harder to pass Gatekeeper. See Resolving Gatekeeper Problems Caused by Dangling Load Command Paths for the details.

Use of an Old macOS SDK

Another dynamic library load failure related to the trusted execution system looks like this:

```
code signature in … '…/LinkToDodo.app/Contents/Frameworks/libDodo.dylib'
not valid for use in process: mapped file has no cdhash, completely
unsigned? Code has to be at least ad-hoc signed.
```

Note The `cdhash` in this message refers to a code directory hash. For more information on cdhashes, see TN3126 Inside Code Signing: Hashes

This is harder to understand, not least because the library *is* actually signed:

```
% codesign -d --vvv LinkToDodo.app/Contents/Frameworks/libDodo.dylib
…
Authority=Apple Development: …
…
```

The explanation can be found tucked away in Notarizing macOS Software Before Distribution, which says:

> Apple's notary service requires you to adopt the following protections:
> …
> - Link against the macOS 10.9 or later SDK

macOS 10.9 introduced important code signing improvements. The hardened runtime depends on those improvements. It confirms their presence by looking at the SDK that the code was built with. If the code was built with an old SDK, or has no record of the SDK it was built with, the hardened runtime refuses to load it.

In this example, the LinkToDodo app was linked to a modern SDK but the `libDodo.dylib` has no record of the SDK it was built with:

```
% vtool -show-build LinkToDodo.app/Contents/MacOS/LinkToDodo
…
        cmd LC_BUILD_VERSION
…
        sdk 12.3
…
% vtool -show-build LinkToDodo.app/Contents/Frameworks/libDodo.dylib
LinkToDodo.app/Contents/Frameworks/libDodo.dylib:
%
```

That explains the error:

1. The process has the hardened runtime enabled.
2. The hardened runtime requires that all code be built with the macOS 10.9 SDK or later.
3. `libDodo.dylib` has no record of the SDK it was build with, so the trusted execution system blocks it from loading.
4. The dynamic linker reports that in its explanation of the problem.

The best fix is to rebuild the code from source with the latest tools. If you can't do that right now, see Notarisation and the macOS 10.9 SDK for a workaround.

IMPORTANT This is a short-term compatibility measure. Plan to rebuild this code from source as soon as possible. If you got the code from another third-party developer, make sure they're aware of this issue.

Finally, if you can only reproduce this problem in the field and have managed to snag a sysdiagnose log of it, look in the system log for a log entry like this:

```
type: default
time: 2022-05-20 13:12:11.185089 +0100
process: kernel
category: <Missing Description>
message: …/LinkToDodo.app/Contents/Frameworks/libDodo.dylib: Possible race detected. Rejecting.
```

That's one cryptic smoking gun!

For general information about the system log, see Your Friend the System Log.

Restricted Entitlements on Library Code

The third dynamic library load failure related to the trusted execution system looks like this:

```
…
OS Version:             macOS 11.6.5 (20G527)
…

Termination Reason:     DYLD, [0x5] Code Signature

Application Specific Information:
dyld: launch, loading dependent libraries

Dyld Error Message:
  Library not loaded: @rpath/OverlyEntitled.framework/Versions/A/OverlyEntitled
  Referenced from: /Users/DSER/vL/AppWithEntitlementLibrary.app/Contents/MacOS/AppWithEntitlementLibrary
  Reason: no suitable image found.  Did find:
    …/AppWithEntitlementLibrary.app/Contents/MacOS/../frameworks/OverlyEntitled.framework/Versions/A/OverlyEntitled: code signature invalid for '…/AppWithEntitlementLibrary.app/Contents/MacOS/../frameworks/OverlyEntitled.framework/Versions/A/OverlyEntitled'
…
```

Note This crash report is from macOS 11. For… well… reasons… macOS 12 ignores entitlements on library code. However, this changed again in macOS 13, which will fail much like macOS 11 did.

However, the code signature is valid:

```
% codesign -v -vvv AppWithEntitlementLibrary.app/Contents/Frameworks/OverlyEntitled.framework
AppWithEntitlementLibrary.app/Contents/Frameworks/OverlyEntitled.framework: valid on disk
AppWithEntitlementLibrary.app/Contents/Frameworks/OverlyEntitled.framework: satisfies its Designated Requirement
```

It also passes both of the tests outlined in the previous section:

```
% codesign -d --vvv AppWithEntitlementLibrary.app
…
TeamIdentifier=S4PME9E2Y8
…
% codesign -d --vvv AppWithEntitlementLibrary.app/Contents/Frameworks/OverlyEntitled.framework
…
TeamIdentifier=S4PME9E2Y8
…
% vtool -show-build AppWithEntitlementLibrary.app/Contents/Frameworks/OverlyEntitled.framework/Versions/A/OverlyEntitled
…
        sdk 12.3
…
```

The issue is that the framework is signed with a restricted entitlement:

```
% codesign -d --entitlements - AppWithEntitlementLibrary.app/Contents/Frameworks/OverlyEntitled.framework
…
[Dict]
  [Key] com.apple.developer.networking.vpn.api
  [Value]
    [Array]
      [String] allow-vpn
```

Entitlements are only effective when applied to a main executable. Code that isn't a main executable is called *library code*, and that includes frameworks, dynamic libraries, and bundles. Do not apply entitlements to library code. At best it's benign. At worse, it causes a code signing crash like this.

Note For details on what constitutes a main executable, see Creating Distribution-Signed Code for Mac

The *Entitlements on macOS* section of TN3125 Inside Code Signing: Provisioning Profiles defines restricted entitlement and makes it clear that, on macOS, every restricted entitlement claimed by an executable must be authorised by its provisioning profile. However, library code does not have an embedded provisioning profile:

- A shared library has no bundle structure, and thus *can't* include a provisioning profile.
- Library code with a bundle structure, frameworks and bundles, *could* have a provisioning profile but most tools, including Xcode, do not embed one.

So, the OverlyEntitled framework is claiming a restricted entitlement but that claim isn't authorised by a profile and thus the trusted execution system tells the dynamic linker not to load it.

To fix this, change your code signing setup so that only main executables claim entitlements. For detailed advice on that topic, see Creating Distribution-Signed Code for Mac.

IMPORTANT The number one cause of this problem is folks signing their code with `--deep`. Don't do that, for this reason and for the other reasons outlined in `--deep` Considered Harmful.

Revision History

- **2022-12-13** Updated the note in the *Restricted Entitlements on Library Code* section to account for macOS 13 "going back to metric".
- **2022-09-26** Fixed a broken link.
- **2022-06-13** Added the *Restricted Entitlements on Library Code* section.
- **2022-05-20** First posted.

Gatekeeper Code Signing Notarization

Reply

Posted 1 year ago by eskimo

Add a Comment