Ship macOS Apps Course

Flavio Copes

Updated August 3, 2026

Contents

Prepare, secure, sign, notarize, package, publish, update, and recover macOS app releases with an honest source-first path.

What you'll learn: Produce a traceable macOS release with a clear security boundary, verified artifact, safe installation path, and tested rollback plan.

Choose a distribution path

Compare source-first, Mac App Store, and direct distribution and inventory the release.

Choose source, App Store, or direct distribution

Select a distribution path from the audience, capabilities, budget, and installation experience instead of treating every Mac app alike.

You built a macOS app. Now you want it on other people's Macs. There are three realistic paths, and the right one depends on your audience, your budget, and how much installation friction you can accept.

Source-first distribution is the free path. You publish the source code, the Xcode version you tested with, and one command that produces a working build. Technical users clone the repository and build the app themselves. No paid Apple Developer Program membership, no certificates, no review queue.

```
git clone https://github.com/flaviocopes/notes
cd notes
xcodebuild -scheme Notes -configuration Release build
```

If your users are developers, this is often all you need. A lot of open source Mac software starts exactly like this.

The **Mac App Store** gives you Apple-hosted downloads, payments, and automatic updates. In exchange you accept App Review, the App Sandbox requirement, and Apple's commission. It fits consumer apps where discovery matters more than control.

Direct distribution means users download a ZIP or DMG from your own site. You control pricing, release timing, and updates. The smooth version of this path needs a paid Apple Developer Program membership (99 USD per year), a **Developer ID Application** certificate, and notarization. With those in place, users double-click your app and it opens. Without them, macOS blocks the first launch.

There is a fourth option for early testing: an unsigned prebuilt app. It works, but be honest about it. Label the download as unsigned and not notarized, publish its SHA-256 checksum, and document Apple's per-app override: **System Settings → Privacy & Security → Open Anyway**.

What you must never do is tell users to disable Gatekeeper globally or strip quarantine attributes from downloads. That advice weakens their entire Mac to route around your missing signature. If installation friction starts hurting you, that's your signal to pay for the membership and notarize.

My advice: start source-first, and add Developer ID signing the moment non-developers want your app.

Identify the app bundle

Inspect the product name, bundle identifier, executable, Info.plist values, and resources that make one macOS app.

Everything you ship on macOS revolves around one artifact: the **app bundle**. Before you sign or package anything, you should know exactly what is inside it.

A `.app` is not a file. It's a directory with a fixed structure that Finder presents as a single application. Look inside your built app:

```
find Notes.app/Contents -maxdepth 2 -print
```

You'll see a predictable layout. `Contents/MacOS` holds the executable. `Contents/Resources` holds assets like images and localizations. `Contents/Frameworks` holds embedded libraries. And `Contents/Info.plist` holds the metadata that describes the app to the system.

Read that metadata with `plutil`:

```
plutil -p Notes.app/Contents/Info.plist
```

The value that matters most is `CFBundleIdentifier`, the **bundle identifier**. Mine is `com.flaviocopes.Notes`. This reverse-DNS string is your app's durable identity. macOS keys preferences, privacy permissions, and Keychain access to it, and Apple's distribution services track your app by it.

Pick it once and never change it. If you rename it later, macOS treats the app as a brand new one: users lose their granted permissions, and their saved preferences point at the old identity.

One more rule, and it will matter a lot in the signing module: once a bundle is signed, treat it as read-only. The signature seals the executable and every resource. Change one file and the seal breaks.

This is a classic failure. You sign the app, then "quickly fix" a typo in a resource file or swap the icon. Now `codesign --verify` fails and Gatekeeper refuses to launch the app on other Macs. The fix is always the same: rebuild, then re-sign the fresh bundle.

Set version and build numbers

Give every release a user-facing version and a monotonically increasing build number that support updates and diagnosis.

Every release you ship needs two numbers, and they answer different questions.

The **version** (`CFBundleShortVersionString`) is the public release number users see, like `1.2.0`. The **build** (`CFBundleVersion`) identifies one exact compiled artifact, like `42`. The version answers "what did we ship?". The build answers "which bytes exactly?".

Read both from a built app:

```
plutil -extract CFBundleShortVersionString raw Notes.app/Contents/Info.plist
plutil -extract CFBundleVersion raw Notes.app/Contents/Info.plist
```

Why two numbers? Because you will build version `1.2.0` more than once. You fix a packaging mistake, rebuild, and now two different artifacts both claim to be `1.2.0`. The build number is what tells them apart. Increment it for every artifact you hand to anyone, including testers.

Apple's tooling can bump the numbers for you. `agvtool` updates every target in one command:

```
xcrun agvtool new-marketing-version 1.2.0
xcrun agvtool next-version -all
```

The first command sets the version. The second increments the build.

Two rules keep you sane later. Never reuse a version-and-build pair for different bytes. And put both numbers in your release records, crash reports, and About window, so any bug report can be traced to one exact artifact.

Here's the failure mode you're preventing. A user reports a crash "in the latest version". You

have three builds of `1.2.0` on disk: one before the crash fix, one after, one from a different branch. Without a distinct build number in the report, you can't tell which one they're running, and you end up debugging the wrong binary.

Check the numbers in the final exported app, not just in Xcode. Build settings and configurations can override what the project editor shows.

Inventory runtime dependencies

List embedded frameworks, helper tools, licenses, external commands, network services, and data migrations before creating a release.

Your release is more than the Swift code you wrote. It may embed frameworks, helper tools, fonts, machine learning models, or command-line binaries. It may also assume things exist on the user's Mac that only exist on yours.

Before packaging anything, take an inventory. Start with the libraries your executable links against:

```
otool -L Notes.app/Contents/MacOS/Notes
```

Read the output carefully. System frameworks under `/System/Library` are fine — every Mac has them. Paths under `/opt/homebrew` or `/usr/local` are a problem: they exist on your machine because you installed something, and they will not exist for your users.

Then find every executable inside the bundle, because each one ships (and later gets signed) as part of your app:

```
find Notes.app/Contents -type f -perm +111 -print
```

For each redistributed component, record where it came from and what license it carries. "I found it in a build folder" is not an answer you want to give later.

If your app runs external commands, decide the strategy explicitly. Either bundle the tool inside the app, or look for it in known locations and degrade gracefully when it's missing, or ask the user to locate it. Never assume `/opt/homebrew/bin` exists on anyone's Mac.

The classic failure looks like this. The app works perfectly for you for months. A user launches it and it dies instantly with `dyld: Library not loaded: /opt/homebrew/lib/libsqlite3.dylib`. Your Homebrew installation was a hidden dependency the whole time.

The test that catches this is cheap: run the release artifact on a Mac (or a fresh macOS virtual machine) that has never seen your development setup. If it launches and every feature works, your inventory is complete.

Check your understanding: choosing a distribution path

Check the key ideas from choosing a distribution path before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Define the security boundary

Configure sandboxing, entitlements, privacy descriptions, and Hardened Runtime.

Enable App Sandbox deliberately

Turn on the sandbox, declare only required capabilities, and verify real behavior rather than checking boxes until errors disappear.

App Sandbox is a containment system. A sandboxed app starts with almost no access: no reading the user's files, no network, no camera. You then declare, capability by capability, exactly what the app needs.

Why accept that restriction? Damage control. If your app is ever compromised — through a malicious file it parses, a bad dependency, anything — the attacker gets your entitlements, not the whole Mac. The Mac App Store requires the sandbox, and it's worth considering for direct distribution too.

Enable it in Xcode under **Signing & Capabilities** by adding the App Sandbox capability. Then add only what the app really needs. For our Notes app, that's reading and writing files the user explicitly picks:

```
<key>com.apple.security.app-sandbox</key>
<true/>
<key>com.apple.security.files.user-selected.read-write</key>
<true/>
```

Notice the shape of that second entitlement. It doesn't grant access to the whole disk. It grants access to files the user selects through an open or save panel. User intent is the permission.

Verify the built app actually carries the sandbox entitlement:

```
codesign -d --entitlements :- Notes.app
```

You can also open Activity Monitor, add the Sandbox column, and confirm the running app shows "Yes".

Then test every real workflow with the release entitlement set: opening files, saving, networking, spawning helpers. Sandbox denials show up as failed operations in the app and as denial messages in Console.

When something is denied, resist the reflex to grant broader entitlements until the error goes away. A denial is information: which resource, which operation, and was there user intent behind it? If the feature genuinely needs the access, add the narrowest capability that unblocks it. If you can redesign the feature around a user-selected file instead, do that.

Read the final entitlements

Inspect the entitlements embedded in the release artifact instead of assuming the Xcode project editor represents the shipped result.

Entitlements are the capabilities granted to your signed code: sandbox on or off, network access, file access, Keychain groups, automation. Xcode shows you what you asked for. The built app is what users actually get, and the two can differ.

Build settings, configurations, and the export step all influence the final entitlement set. So the habit to build is: inspect the artifact, not the project.

```
codesign -d --entitlements :- Notes.app
```

The `:-` writes the embedded entitlements to standard output. Read the list slowly. Every entry

should map to a feature you can name. Entitlements left over from an abandoned experiment are pure attack surface — delete them from the project and rebuild.

Watch for temporary exception entitlements in particular. They exist for migration scenarios, and a final build carrying broad exceptions is a design smell.

One entitlement deserves special attention: `com.apple.security.get-task-allow`. Debug builds carry it so the debugger can attach. A distributed build must not. Check for it explicitly:

```
codesign -d --entitlements :- Notes.app | grep get-task-allow
```

No output is the answer you want. If it's there, you exported a debug-configured build — and notarization will reject it, with exactly that entitlement named in the log.

Nested code has its own entitlements. Embedded frameworks and helper executables are signed separately, so inspect them too:

```
codesign -d --entitlements :- \
  Notes.app/Contents/Frameworks/Sparkle.framework
```

A helper with broader capabilities than the main app undermines the boundary you designed.

Finish with a one-line rule: for every entitlement in the shipped artifact, you can say which feature needs it and why. If you can't, remove it.

Explain privacy-sensitive access

Add accurate usage descriptions and request protected resources only when the user activates the feature that needs them.

macOS guards a set of resources it considers privacy-sensitive: camera, microphone, location, contacts, calendars, screen recording, and more. Your app can't touch them silently. The first access triggers a system permission prompt, and the user decides.

Your part of that prompt is the **purpose string**: an entry in `Info.plist` that explains why you're asking. If our Notes app records voice memos, it needs one for the microphone:

```
<key>NSMicrophoneUsageDescription</key>
<string>Notes records audio when you attach a voice memo to a note.</string>
```

Write it like that: name the visible feature, say what happens with the data. "App needs microphone access" tells the user nothing and earns you denials.

The purpose string is not optional. If your code touches a protected resource and the matching key is missing, macOS kills the app on the spot. The crash report says the app accessed privacy-sensitive data without a usage description. That one is confusing the first time you hit it, because everything worked until the exact line that touched the microphone.

Timing matters as much as wording. Ask in context: request microphone access when the user taps record for the first time, not at launch next to three other prompts. A request tied to a visible action gets approved far more often.

Then test the paths developers skip. First request. Denial — the app must keep working with the feature disabled gracefully, and point the user to System Settings if they change their mind. Re-test the first-run experience by resetting the permission:

```
tccutil reset Microphone com.flaviocopes.Notes
```

Never loop the prompt or treat a denial as an error to retry. The user answered. Respect it.

Enable Hardened Runtime

Adopt runtime protections for distributed code and grant exceptions only for behavior the application can justify and test.

The **Hardened Runtime** is a set of protections your app opts into at signing time. It blocks code injection, loading of unsigned or tampered libraries, and debugger attachment in shipped builds. You're telling macOS: this process should never be modified at runtime — enforce that.

You need it. Notarization requires the Hardened Runtime on every executable in a Developer ID distribution. No hardened runtime, no notarization, no smooth installs.

In Xcode, add the Hardened Runtime capability under **Signing & Capabilities** for the app target and every helper or bundled tool target. If you sign manually in a script, it's the `--options runtime` flag:

```
codesign --force --options runtime \
  --sign "Developer ID Application: Flavio Copes (ABCDE12345)" \
  Notes.app
```

Verify it took. The flags line must include `runtime`:

```
codesign -d --verbose=2 Notes.app
```

Look for `flags=0x10000(runtime)` in the output.

The runtime has escape hatches: exceptions for just-in-time compilation, unsigned executable memory, debugging, and library validation. Some apps genuinely need one — an app embedding a JavaScript engine with JIT, for example. But every exception switches off one specific protection, so treat them like sandbox entitlements: none by default, each one justified by a feature you can name.

When a third-party dependency crashes under the hardened runtime, investigate before reaching for an exception. Often the real problem is a badly signed framework, and the right fix is re-signing or updating the dependency, not disabling library validation for your whole app.

And check that debug entitlements are gone. `com.apple.security.get-task-allow` must not survive into a shipping build.

The failure you'll actually meet: a scripted build that signs without `--options runtime`. Everything works locally. Then `notarytool` returns `Invalid`, and the log says the executable does not have the hardened runtime enabled. Recognize that message, add the flag, re-sign, resubmit.

Check your understanding: defining the security boundary

Check the key ideas from defining the security boundary before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Sign and notarize

Archive, sign with the correct identity, notarize, staple, and verify the exact app.

Understand signing identities

Distinguish development, Developer ID, and Mac App Store identities and verify which identity signed a release.

Code signing answers one question for macOS: who produced these bytes, and have they changed since? The answer comes from a **signing identity** — a certificate plus its private key — and different identities mean different things.

Apple Development identities are for your own Macs. They let you build, run, and debug. Ship a build signed with one and other Macs will refuse it.

Developer ID Application is the identity for direct distribution outside the Mac App Store. Gatekeeper recognizes it, and it's the identity notarization expects. Getting one requires the paid Apple Developer Program membership.

Mac App Store submissions use Apple's store certificates and submission flow — a separate path where Apple re-signs your app for delivery.

There is also the **ad-hoc signature**: signing with `-` instead of an identity. Apple silicon requires every binary to carry at least an ad-hoc signature to run at all. But an ad-hoc signature carries no identity. It proves nothing to anyone else's Mac, and Gatekeeper treats the app as unsigned.

See which identities your Mac can sign with:

```
security find-identity -v -p codesigning
```

You want a line containing `Developer ID Application: Flavio Copes (ABCDE12345)`.

Then read the identity from a built artifact, which is the fact that matters:

```
codesign -dv --verbose=4 Notes.app
```

Check the `Authority` chain — it should start with your Developer ID certificate — and the `TeamIdentifier`.

The private key behind your certificate is the crown jewel. Anyone holding it can ship malware in your name. Never commit an exported certificate or its password, and share it only through a proper secrets manager.

The everyday mistake: Xcode automatic signing quietly used the development certificate, you zipped the result, and a tester reports the app "is damaged or can't be opened". Run `codesign -dv` on your artifact before shipping and you catch it in one second.

Archive a release build

Create an Xcode archive from a clean release configuration and preserve the archive as the input to export and notarization.

An **Xcode archive** is the frozen output of one release build: the app, its debug symbols, and metadata about how it was produced. Everything downstream — export, signing, notarization, crash symbolication — derives from it.

In Xcode the flow is **Product → Archive**, which builds the release configuration and opens the

result in Organizer. From a script, it's two `xcodebuild` steps. First, archive:

```
xcodebuild -project Notes.xcodeproj \
  -scheme Notes \
  -configuration Release \
  archive -archivePath build/Notes.xcarchive
```

Then export a Developer ID-signed app from the archive. The export step reads a small plist that names the distribution method:

```
<key>method</key>
<string>developer-id</string>
```

```
xcodebuild -exportArchive \
  -archivePath build/Notes.xcarchive \
  -exportOptionsPlist ExportOptions.plist \
  -exportPath build/export
```

The signed app lands in `build/export/Notes.app`.

Before distributing, inspect what you got: the version and build, the signing identity, the entitlements, the embedded frameworks. Fix problems by changing the project and archiving again. Never edit the exported app by hand — you learned why in the bundle lesson. Edits break the seal.

Keep the archive. It contains the `dSYM` debug symbol bundles that turn the addresses in a user's crash report back into function names and line numbers. Lose the archive and that crash report from version 1.2.0 becomes hexadecimal soup.

Record alongside it the source commit, the Xcode version, and the export options. The archive is the bridge between "this commit" and "these shipped bytes", and the record is what makes the bridge usable months later.

A subtle failure: archiving with uncommitted local changes. The archive says 1.2.0, your repository says 1.2.0, but they differ by the three lines you never committed. Archive from a clean working tree, ideally in CI.

Submit to the notary service

Send a Developer ID-signed archive to Apple, wait for the result, and read the complete log before treating acceptance as success.

Notarization is Apple's automated malware scan for software distributed outside the Mac App Store. You upload a signed artifact, Apple's service scans it, and on success it issues a **ticket** recording that this exact software passed. Gatekeeper looks for that ticket at first launch.

Understand what it is not. Notarization is not App Review. No human looks at your app, nobody judges the UI, and acceptance doesn't prove the app works. It proves Apple scanned these bytes and found no known malware.

The command-line tool is `notarytool`, bundled with Xcode. The older `altool` upload path is dead — Apple no longer accepts it. Xcode's Organizer can also notarize for you, but a scripted workflow gives you the same steps reproducibly.

First, store credentials once in a Keychain profile. Use your Apple ID with an app-specific password, plus your team ID:

```
xcrun notarytool store-credentials "notes-notary" \
  --apple-id "flavio@flaviocopes.com" \
  --team-id ABCDE12345
```

Then submit a ZIP, DMG, or installer package and wait for the verdict:

```
xcrun notarytool submit Notes-1.2.0.zip \
  --keychain-profile "notes-notary" \
  --wait
```

With `--wait`, the command polls until processing finishes and prints the result. `status: Accepted` is what you want. Typical waits are a few minutes.

When you get `status: Invalid` instead, don't guess. Every submission has an identifier, and the service keeps a detailed log. Fetch it:

```
xcrun notarytool log 2efe2717-52ef-43a5-96dc-0797e4ca1041 \
  --keychain-profile "notes-notary"
```

The JSON lists each issue with the offending file path. The usual suspects: an executable without the hardened runtime, a binary signed without a secure timestamp, or an unsigned helper buried inside a framework you embedded.

Two habits for automation. Capture the submission identifier in your build logs so failures stay diagnosable later. And never print your Apple ID password or API keys — that's what the Keychain profile is for.

Staple and verify the ticket

Attach the notarization ticket where supported and verify signatures, Gatekeeper assessment, and stapling on the exact artifact you will publish.

Notarization produced a ticket, but the ticket lives on Apple's servers. **Stapling** attaches a copy to your artifact, so Gatekeeper can verify the app even when the user's Mac is offline or Apple's servers are unreachable at launch time.

Staple to the app (or to the DMG, if that's your deliverable), then confirm the staple took:

```
xcrun stapler staple Notes.app
xcrun stapler validate Notes.app
```

`staple` prints a success message, and `validate` confirms the ticket is readable from the artifact itself.

Now verify the whole chain the way Gatekeeper will. Two checks, two different questions. First, is the signature intact on every nested component?

```
codesign --verify --deep --strict --verbose=2 Notes.app
```

Second, would macOS actually let this app run?

```
spctl -a -vv Notes.app
```

The output you want ends with `accepted` and `source=Notarized Developer ID`. If it says `rejected`, stop — users will see a block dialog, and you need the reason before shipping.

One clarification on `--deep`, because it trips people up. It's a verification convenience that walks nested code. It is not a signing strategy. Sign nested components correctly from the inside out

(Xcode's export does this for you) and use `--deep` only to catch mistakes.

Run these checks twice: on the artifact before packaging, and again on an app extracted from the final download. That second run catches a classic ordering bug: you zipped the app, submitted the ZIP for notarization, stapled the app afterwards — but the ZIP you published still contains the unstapled copy. Offline Macs then get a worse first launch than the one you tested. Staple first, package second.

Check your understanding: signing and notarization

Check the key ideas from signing and notarization before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Package and publish

Create ZIP and DMG artifacts, publish checksums, and document unsigned previews safely.

Create a ZIP with ditto

Package an application while preserving macOS bundle metadata and verify the extracted copy before publishing.

A ZIP is the lightest way to ship a direct download. But an app bundle is a directory full of symlinks, extended attributes, and structure that generic ZIP tools can mangle. On macOS, the tool that gets it right is `ditto`.

Create the archive from the signed, stapled app:

```
ditto -c -k --sequesterRsrc --keepParent \
  Notes.app Notes-1.2.0.zip
```

`-c -k` means create a PKZip archive. `--keepParent` puts `Notes.app` itself in the archive, so extraction yields the bundle rather than its loose contents. `--sequesterRsrc` preserves resource metadata the way Archive Utility expects.

Why not `zip -r`? Frameworks inside your app contain symlinks (`Versions/Current` and friends). A tool that stores those as duplicate files breaks the bundle structure, and with it the code signature. The app you tested is valid; the app users extract is not.

So the test that matters happens after extraction. Unpack into a fresh directory and verify the extracted copy:

```
ditto -x -k Notes-1.2.0.zip extracted/
codesign --verify --deep --strict extracted/Notes.app
spctl -a -vv extracted/Notes.app
```

Then launch it, check the version in the About window, and walk through the first-run permissions. This extracted copy is what users run — it's the thing to test, not the app in your build folder.

Two packaging rules to close. Ship one archive, not a folder of loose files: one artifact, one checksum, one installation instruction. And name the file with the public version — `Notes-1.2.0.zip` — never with secrets, usernames, or machine-specific build paths that would leak into public URLs.

Create and test a disk image

Build a simple read-only DMG, place the signed app inside it, and verify both the disk image and copied application.

A **disk image** is the classic Mac install experience: the download mounts as a volume, the window shows your app next to an Applications alias, and the user drags the app across. One gesture, no installer.

Build the image from a staging folder containing exactly what users should see — the app and an `Applications` symlink:

```
mkdir release-dmg
cp -R Notes.app release-dmg/
ln -s /Applications release-dmg/Applications
```

Then create a compressed, read-only image:

```
hdiutil create -volname "Notes" \
  -srcfolder release-dmg \
  -ov -format UDZO Notes-1.2.0.dmg
```

`UDZO` is the compressed read-only format meant for distribution. Read-only is a feature here: nothing inside the image can change after you verify it.

If the DMG is what you notarize and publish, staple the ticket to the DMG itself:

```
xcrun stapler staple Notes-1.2.0.dmg
xcrun stapler validate Notes-1.2.0.dmg
```

Now test it the way a user experiences it. Mount the image, drag the app to Applications, eject the image, and launch the copy from `/Applications`. Do this on a standard (non-admin) account.

Why insist on the copy step? Apps launched straight from a mounted image run from a read-only, temporary location — macOS may even run them translocated from a randomized path. Users who skip the drag end up with an app that vanishes when the image ejects, or one that can't write its own support files. Make the drag target obvious in the window, and never require writes inside the read-only image.

My advice on appearance: skip the custom background art and pixel-perfect icon layouts, at least at first. Fancy DMG layouts are a pile of fragile build steps for a window users see once.

Publish checksums and release notes

Give each download an integrity hash, exact version, requirements, changes, known issues, and installation instructions.

Two small things separate a professional release from a bare file link: an integrity checksum and honest release notes.

The **SHA-256 checksum** is a fingerprint of the exact bytes you published. Compute it from the final file — the one you actually upload, after any re-zipping or renaming:

```
shasum -a 256 Notes-1.2.0.zip
```

Publish the filename and the hash right next to the download link. A user (or your future self) verifies a download by recomputing it:

```
shasum -a 256 ~/Downloads/Notes-1.2.0.zip
```

The two hex strings must match exactly. A mismatch means the bytes differ — a corrupted download, a stale mirror, or a tampered file — and the right response is to not run it.

Be clear about what a checksum is and isn't. It detects corruption and lets careful users confirm they received what you published. It does not establish who you are — that's the job of code signing. You want both.

The sneaky failure: you compute the hash, then notice a typo in the filename, re-create the archive, and upload the new one. Compression timestamps differ, so the published hash no longer matches the published file — and every careful user now thinks your release is compromised. Hash last, upload immediately, then verify the uploaded file once more.

Release notes are the other half. State the version and build, the minimum macOS version, supported architectures, what changed, what you fixed, any data migrations, and known issues. "Bug fixes and improvements" helps nobody.

If you offer multiple artifacts, write separate installation instructions for each: source builds, the signed and notarized download, and any unsigned preview. Never blur the line between them — an unsigned artifact must never be described in a way that implies Apple notarized it.

Document an unsigned preview safely

Offer an optional unsigned build without hiding its limitations or weakening the user's Mac security settings.

Sometimes you want feedback on a prebuilt app before paying for the Apple Developer Program. That's a legitimate stage. There is an honest way to do it and a harmful way, and this lesson is about staying on the honest side.

It starts with labeling. Right next to the download link, say it plainly: this build is **unsigned and not notarized**. State the source commit it was built from, the build requirements, and what the preview is meant to test. Publish the SHA-256 checksum:

```
shasum -a 256 Notes-0.9.0-preview.zip
```

Then document what testers will experience, because you should be the one explaining it — not a confusing error dialog. On first launch, macOS reports it can't verify the app is free of malware and blocks it. You can see the same verdict yourself:

```
spctl -a -vv Notes.app
```

For an unsigned build it prints `rejected`. That's Gatekeeper doing its job, not a bug.

Apple provides a per-app override for exactly this situation. The tester tries to open the app once, then goes to **System Settings → Privacy & Security**, scrolls to the message about the blocked app, and clicks **Open Anyway**. macOS asks for confirmation, and from then on that one app opens normally.

That flow is deliberately narrow: one user decision, one app, everything else still protected. Which is why the alternatives floating around the internet are unacceptable. Never instruct users to disable Gatekeeper system-wide or to strip the quarantine attribute from downloads with terminal commands. Those recipes turn off protections for everything, to save one click, and they train people to paste security-weakening commands from strangers.

Keep the source-first build instructions published beside the preview, so technical users can build instead of trusting binaries.

And treat friction as a signal. When Open Anyway explanations start filling your support inbox, that's the moment paid Developer ID signing and notarization pay for themselves.

Check your understanding: packaging and publishing

Check the key ideas from packaging and publishing before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate releases

Test clean installation, design updates and rollback, protect automation, and run the full checklist.

Test on a clean standard account

Run the final package outside the developer environment and verify installation, permissions, storage, updates, and removal.

Your Mac is the worst place to judge a release. It has your certificates, your Homebrew packages, your granted permissions, and years of preferences. A release that works there proves almost nothing. The boundary that matters is a Mac that has never met you.

The cheap version of that boundary is a **clean standard account**. Create a new standard (non-admin) user in System Settings, log into it, and act like a customer.

Start from the public URL — the real one, not a local file copy:

```
curl -LO https://flaviocopes.com/downloads/Notes-1.2.0.zip
shasum -a 256 Notes-1.2.0.zip
```

Confirm the hash matches what you published. Then extract, and check what Gatekeeper thinks of the result:

```
ditto -x -k Notes-1.2.0.zip .
spctl -a -vv Notes.app
```

You want `accepted` with `source=Notarized Developer ID`.

Now run the human part of the test. First launch: no scary dialogs beyond the expected permission prompts. Each prompt shows your purpose string and appears at a sensible moment. Create some data, quit, relaunch, and confirm it persisted. Exercise every feature that touches an external dependency.

What makes the clean account powerful is what it lacks. No development certificates. No `/opt/homebrew` safety net. No privacy permissions you approved months ago and forgot about. Missing dependencies and missing purpose strings that hide on your account fail loudly here.

It has limits, though. The clean account still shares your macOS version and any system-wide software. For releases that matter, repeat the test on a fresh macOS virtual machine or a second Mac.

Close the loop by recording the result: macOS version, architecture, artifact hash, pass or fail, and anything odd. When a user later reports a problem, that record tells you instantly whether they're hitting something you tested or something new.

Design updates and rollback

Decide how users discover releases, preserve data compatibility, and return to a previous version when an update fails.

Shipping 1.0 is the start of a loop. Users need a way to learn about new versions, get them safely, and — when an update goes wrong — get back to a version that works.

Match the mechanism to your distribution path. A source-first project can announce releases on the repository and keep tagged versions buildable. A direct-download app can document a manual flow (a "Check for Updates" menu item that opens the releases page), or integrate an update framework.

The **Sparkle-style** model is the standard for direct distribution: the app periodically fetches an appcast feed listing versions and download URLs, and every update is cryptographically signed with a key only you hold. The app verifies the signature before installing. Two properties are non-negotiable whatever tool you use: the update channel is HTTPS, and the update verifies origin and integrity before replacing anything. An unsigned update channel is a remote code execution service with your app's name on it.

Whatever the mechanism, every published update gets the same treatment as a first install — notarized, stapled, and checksummed:

```
shasum -a 256 Notes-1.3.0.zip
```

Updates ship with data. If version 1.3.0 migrates the notes database, make a backup first and record a format version inside the data store. My rule: decide the supported rollback window, then keep every version in that window able to read the current format — or keep the pre-migration backup restorable.

Test the ugly paths, because updates fail in the field constantly: a download interrupted halfway, no network, a disk nearly full, and a user jumping from 1.0.2 straight to 1.4.0 past three migrations.

Write the rollback plan down before you need it: which previous artifact stays downloadable, whether it can read data written by the current version, and the exact steps to restore a migration backup.

The failure that teaches this lesson: 1.3.0 migrates the database, a crash appears, and users reinstall 1.2.0 — which opens the migrated file and silently corrupts it. The old binary alone is not a rollback plan. The old binary plus compatible data is.

Protect release automation

Keep signing and notarization credentials out of untrusted jobs and make every automated artifact traceable to reviewed source.

The moment releases run in CI, your pipeline holds real power: a certificate that speaks in your name and credentials that talk to Apple. Protect that job differently from ordinary tests.

The first rule is separation. Pull-request jobs run tests on unreviewed code — code anyone can propose. They must never see signing keys or notarization credentials, or a malicious pull request can exfiltrate them. Expose release secrets only to protected workflows: a protected tag, a manually approved environment, a release branch with review requirements.

Store secrets in the platform's secret store, never in the repository. On the runner, import the certificate into a dedicated temporary keychain:

```
security create-keychain -p "$KEYCHAIN_PASSWORD" build.keychain
security import certificate.p12 -k build.keychain \
  -P "$CERT_PASSWORD" -T /usr/bin/codesign
security unlock-keychain -p "$KEYCHAIN_PASSWORD" build.keychain
```

And never echo secret values. A single `set -x` in the wrong script prints your certificate password into a build log that outlives the job.

Here's the CI failure everyone meets once: `codesign` dies with `errSecInternalComponent`. The message is useless, but the cause is almost always a locked keychain — headless runners don't get the unlock that logging in normally provides. The `security unlock-keychain` call above, plus `security set-key-partition-list` to let `codesign` use the key without a UI prompt, is the fix.

Pin the toolchain. Select the exact Xcode version the release needs and pin dependency versions, so the same tag produces the same bytes next month.

Then make every artifact traceable. Record the source commit, the workflow run, the version and build, the artifact's SHA-256, the signing identity, and the notarization submission identifier.

Keep one human in the loop. Automation should assemble the release; a person reviews the version and the notes and presses publish. Automation removes repetition, not accountability.

Run the complete release checklist

Produce one traceable macOS release whose source, archive, signature, package, verification, notes, and recovery path agree.

You now have every piece. This lesson chains them into one run, because a release is exactly that: one traceable chain from a commit to a verified download.

Start by freezing. Pick the release commit, set the version and build, and commit that change. From here, everything must come from this state — no "one more quick fix" between archive and upload.

Build and inspect:

```
xcodebuild -project Notes.xcodeproj -scheme Notes \
  -configuration Release \
  archive -archivePath build/Notes.xcarchive
xcodebuild -exportArchive -archivePath build/Notes.xcarchive \
  -exportOptionsPlist ExportOptions.plist -exportPath build/export
```

Check the exported app's version, entitlements, and signing identity before going further. Catching a debug entitlement now costs a minute; catching it after notarization costs a full round trip.

Notarize, staple, then package the stapled app — in that order:

```
ditto -c -k --sequesterRsrc --keepParent \
  build/export/Notes.app Notes-1.2.0.zip
xcrun notarytool submit Notes-1.2.0.zip \
  --keychain-profile "notes-notary" --wait
xcrun stapler staple build/export/Notes.app
```

Remember the ordering rule from the stapling lesson: after stapling the app, rebuild the ZIP from the stapled copy so the published archive contains the ticket. Then verify the final artifact:

```
shasum -a 256 Notes-1.2.0.zip
codesign --verify --deep --strict build/export/Notes.app
spctl -a -vv build/export/Notes.app
```

For a source-first release, the equivalent chain is shorter but just as real: tag the commit, verify the documented clean build on a fresh checkout, and label any optional unsigned artifact accurately.

Publish the artifact with its checksum, system requirements, release notes, installation steps, data migration notes, and rollback instructions. Then become a user: download from the public URL on a clean standard account and repeat the smoke test.

Finish with the release record: commit, archive path, version, build, artifact hash, signing identity, notarization submission identifier, and test results. The standard you're aiming for fits in one sentence — for any release, you can answer which source produced the bytes, who signed them, what Apple accepted, and how a user gets back to a working version.

Check your understanding: operating releases

Check the key ideas from operating releases before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/ship-macos-apps/.