

# Expanding Apple Ecosystem Access with Open Source, Multi Platform Code Signing

A little over one year ago, I [announced a project to implement Apple code signing in pure Rust](#). There have been quite a number of developments since that post and I thought a blog post was in order. So here we are!

But first, some background on why we're here.

## Background

(Skip this section if you just want to get to the technical bits.)

Apple runs some of the largest and most profitable software application ecosystems in existence. Gaining access to these ecosystems has traditionally required the use of macOS and membership in the Apple Developer Program.

For the most part this makes sense: if you want to develop applications for Apple operating systems you will likely utilize Apple's operating systems and Apple's official tooling for development and distribution. Sticking to the paved road is a good default!

But many people want more... flexibility. Open source developers, for example, often want to distribute cross-platform applications with minimal effort. There are entire programming language ecosystems where the operating system you are running on is abstracted away as an implementation detail for many applications. **By creating a de facto requirement that macOS, iOS, etc development require the direct access to macOS and (often above market priced) Apple hardware, the distribution requirements imposed by Apple's software**

## **ecosystems are effectively exclusionary and prevent interested parties from contributing to the ecosystem.**

One of the aspects of software distribution on Apple platforms that trips a lot of people up is code signing and notarization. Essentially, you need to:

1. Embed a cryptographic signature in applications that effectively attests to its authenticity from an Apple Developer Program associated account. (This is signing.)
2. Upload your application to Apple so they can inspect it, verify it meets requirements, likely store a copy of it. Apple then issues their own cryptographic signature called a *notarization ticket* which then needs to be *stapled*/attached to the application being distributed so Apple operating systems can trust it. (This is notarization.)

Historically, these steps required Apple proprietary software run exclusively from macOS. This means that even if you are in a software ecosystem like Rust, Go, or the web platform where you can cross-compile apps without direct access to macOS (testing is obviously a different story), you would still need macOS somewhere if you wanted to sign and notarize your application. And signing and notarization is effectively required on macOS due to default security settings. On mobile platforms like iOS, it is impossible to distribute applications that aren't signed and notarized unless you are running a jailbroken device.

A lot of people (myself included) have grumbled at these requirements. Why should I be forced to involve an Apple machine as part of my software release process if I don't need macOS to build my application? Why do I have to go through a convoluted dance to sign and notarize my application at release time - can't it be more streamlined?

When I looked at this space last year, I saw some obvious inefficiencies and room to improve. So as I said then, I *foolishly* set out to reimplement Apple code signing so developers would have more flexibility and

opportunity for distributing applications to Apple's ecosystems.

**The ultimate goal of this work is to expand Apple ecosystem access to more developers.** A year later, I believe I'm delivering a product capable of doing this.

## One Year Later

**Foremost, I'm excited to announce release of [rcodesign 0.14.0](#). This is the first time I'm publishing pre-built binaries (Linux, Windows, and macOS) of `rcodesign`. This reflects my confidence in the relative maturity of the software.**

In case you are wondering, yes, the macOS `rcodesign` executable is self-signed: it was signed by a GitHub Actions Linux runner using a code signing certificate exclusive to a YubiKey. That YubiKey was plugged into a Windows 11 desktop next to my desk. The `rcodesign` executable was not copied between machines as part of the signing operation. Read on to learn about the sorcery that made this possible.

A lot has changed in the [apple-codesign](#) project / Rust crate in the last year! Just look at the [changelog](#)!

The project was renamed from `tugger-apple-codesign`.

(If you installed via `cargo install`, you'll need to `cargo install --force apple-codesign` to force Cargo to overwrite the `rcodesign` executable with one from a different crate.)

The `rcodesign` CLI executable is still there and more powerful than ever. You can still sign Apple applications from Linux, Windows, macOS, and any other platform you can get the Rust program to compile on.

There is now [Sphinx documentation for the project](#). This is published on

readthedocs.io alongside PyOxidizer's documentation (because I'm using a monorepo). There's some general documentation in there, such as a guide on how to [selectively bypass Gatekeeper](#) by deploying your own alternative code signing PKI to parallel Apple's. (This seems like something many companies would want but for whatever reason I'm not aware of anyone doing this - possibly because very few people understand how these systems work.)

There are bug fixes galore. When I look back at the state of `rcodesign` when I first blogged about it, I think of how naive I was. There were a myriad of applications that wouldn't pass notarization because of a long tail of bugs. There are still known issues. But **I believe many applications will successfully sign and notarize now**. I consider failures novel and worthy of bug reports - so please [report them](#)!

Read on to learn about some of the notable improvements in the past year (many of them occurring in the last two months).

## Support for Signing Bundles, DMGs, and .pkg Installers

When I announced this project last year, only Mach-O binaries and trivially simple `.app` bundles were signable. And even then there were a ton of subtle issues.

`rcodesign sign` can now sign more complex bundles, including many nested bundles. There are reports of iOS app bundles signing correctly! (However, we don't yet have good end-user documentation for signing iOS apps. I will gladly accept PRs to improve the documentation!)

The tool also gained support for signing `.dmg` disk image files and `.pkg` flat package installers.

Known limitations with signing are now [documented](#) in the Sphinx docs.

**I believe `rcodesign` now supports signing all the major file formats used for Apple software distribution.** If you find something that doesn't sign and it isn't documented as a known issue with an existing GitHub issue tracking it, please report it!

## Support for Notarization on Linux, Windows, and macOS

Apple publishes a Java tool named [Transporter](#) that enables you to upload artifacts to Apple for notarization. They make this tool available for Linux, Windows, and of course macOS.

While this tool isn't open source (as far as I know), usage of this tool enables you to notarize from Linux and Windows while still using Apple's official tooling for communicating with their servers.

`rcodesign` now has support for invoking Transporter and uploading artifacts to Apple for notarization. We now support notarizing bundles, `.dmg` disk images, and `.pkg` flat installer packages. I've successfully notarized all of these application types from Linux.

(I'm capable of implementing an alternative uploader in pure Rust but without assurances that Apple won't bring down the ban hammer for violating terms of use, this is a bridge I'm not yet willing to cross. The requirement to use Transporter is literally the only thing standing in the way of making `rcodesign` an all-in-one single file executable tool for signing and notarizing Apple software and I **really** wish I could deliver this user experience win without reprisal.)

**With support for both signing and notarizing all application types, it is now possible to release Apple software without macOS involved in your release process.**

## YubiKey Integration

I try to use my YubiKeys as much as possible because a secret or private key stored on a YubiKey is likely more secure than a secret or private key sitting around on a filesystem somewhere. If you hack my machine, you can likely gain access to my private keys. But you will need physical access to my YubiKey and to compel or coerce me into unlocking it in order to gain access to its private keys.

**rcodesign now has support for using YubiKeys for signing operations.**

This does require an off-by-default smartcard Cargo feature. So if building manually you'll need to e.g. `cargo install --features smartcard apple-codesign`.

The YubiKey integration comes courtesy of the amazing [yubikey](#) Rust crate. This crate will speak directly to the smartcard APIs built into macOS and Windows. So if you have an `rcodesign` build with YubiKey support enabled, YubiKeys should *just work*. Try it by plugging in your YubiKey and running `rcodesign smartcard-scan`.

YubiKey integration has its [own documentation](#).

I even implemented some commands to make it easy to manage the code signing certificates on your YubiKey. For example, you can run `rcodesign smartcard-generate-key --smartcard-slot 9c` to generate a new private key directly on the device and then `rcodesign generate-certificate-signing-request --smartcard-slot 9c --csr-pem-path csr.pem` to export that certificate to a Certificate Signing Request (CSR), which you can exchange for an Apple-issued signing certificate at [developer.apple.com](https://developer.apple.com). **This means you can easily create code signing certificates whose private key was generated directly on the hardware device and can never be exported.** Generating keys this way is widely considered to be more secure than storing keys in software vaults, like Apple's Keychains.

# Remote Code Signing

The feature I'm most excited about is what I'm calling [remote code signing](#).

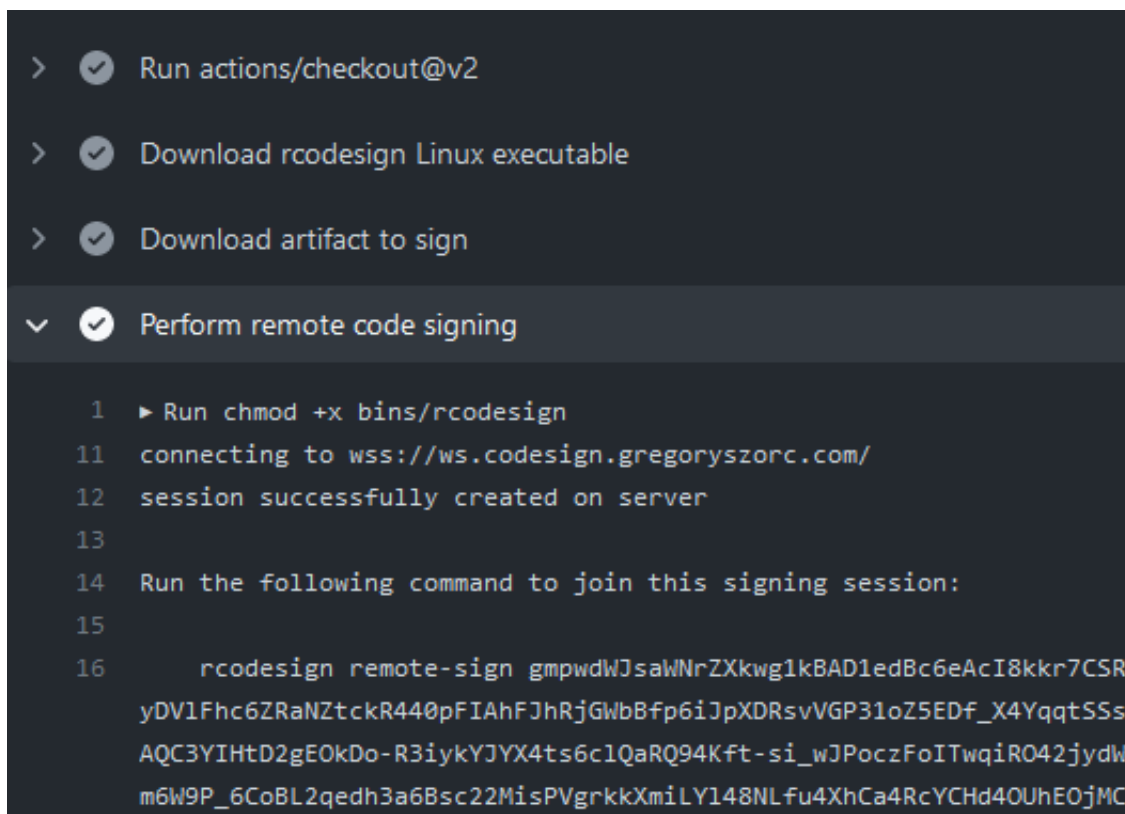
Remote code signing allows you to delegate the low-level cryptographic signature operations in code signing to a separate machine.

It's probably easiest to just demonstrate what it can do.

**Earlier today I signed a macOS universal Mach-O executable from a GitHub-hosted Linux GitHub Actions runner using a YubiKey physically attached to the Windows 11 machine next to my desk at home. The signed application was not copied between machines.**

Here's how I did it.

I have a GitHub Actions workflow that calls `rcodesign sign --remote-signer`. I manually triggered that workflow and started watching the near real time job output with my browser. Here's a screenshot of the job logs:



```
> ✓ Run actions/checkout@v2
> ✓ Download rcodesign Linux executable
> ✓ Download artifact to sign
▼ ✓ Perform remote code signing
  1 ▶ Run chmod +x bins/rcodesign
 11 connecting to wss://ws.codesign.gregoryszorc.com/
 12 session successfully created on server
 13
 14 Run the following command to join this signing session:
 15
 16     rcodesign remote-sign gmpwdWJsawNrZXkwg1kBAD1edBc6eAcI8kkr7CSR-
yDV1Fhc6ZRaNZtckR440pFIAhFJhRjGwbBfp6iJpXDRsvVGP31oZ5EDf_X4YqqtSSs
AQC3YIhtD2gEOkDo-R3iykYJYX4ts6clQaRQ94Kft-si_wJPoczFoITwqiRO42jydWd
m6W9P_6CoBL2qedh3a6Bsc22MisPVgrkkXmiLY148NLfu4XhCa4RcYCHd40UhEOjMC
```



```
    RWTccgDmKTxqHar5R31dp-KnJOPRZpce9ZuUpjjBTixZmsHz-gDvjcPz5LkUw62ZC1o
17
18 Or if this output is too long, paste the following output:
19
20 -----BEGIN SESSION JOIN STRING-----
21 gmpwdWJsawNrZXkwg1kBAD1edBc6eAcI8kkr7CSR+34Kg0ubiW3bkW/bmef3pZqe
22 iYRmgvGRfv28x1p35tpjossoxxRcyjwyowXk012d7CxiizMw5l/PmaIn2Z13ub91
23 dIwltbpGo1SPpWuGC+SVr1xiRK39e91SdnEN+yDVlFhc6ZRaNZtckR440pFIAhFJ
24 hRjGWbBfp6iJpXDRsvVGP31oZ5EDf/X4YqqtSSsfdrERj9F6bmaR4YSc0j61sJT
25 RLv2XHlou1EA5MCm2qvKBZuPU3DHcv3ZShf7FHUa2IbePlawJjwFKE7BrnccCvDN
26 e/7JJwRTkgONsVXtApx42Sw8XE6B7fryZvIx1Jrzy45ZASYwggEiMA0GCSqGSIB3
27 DQEBAQUAA4IBDwAwggEKAoIBAQC3YIHtD2gEOKDo+R3iykYJYX4ts6c1QaRQ94Kf
28 t+si/wJPoczFoITwqiRO42jydWolb05zL+gUc7ZKInQfYQwWBpYhWB+Im5kogzDM
29 PMgS2BXjxUT1X70BFojkk4Wb5u6K8C2UJ1P0lQBKxwipVvXTQurJ/ygB3rv1vLCz
30 ETKKBVPEAVD6ocV/KTT8Pxny2Dt76iyhcAWgf8Tp90ppQy8RTCgy+m6W9P/6CoBL
31 2qedh3a6Bsc22MisPVgrkkXmily148NLfu4XhCa4RcYCHd40UheOjMCoLQ56DTM9
32 W4UEVnLWJXLg1Lxphj02JooFcQwNWTPTvvOfD9wmn37Ad1ZAgMBAAFY24/HV0CB
33 gPl++soH+cU82atz2H+Ug9NQ9sGvI7JgNJgF0bPNbCsSbM7vUKfn2gKT2bNbMgKR
34 9uWTMdkwIxKnbsWaE42kWBdYt2u+RWTccgDmKTxqHar5R31dp+KnJOPRZpce9ZuU
35 pjjBTixZmsHz+gDvjcPz5LkUw62ZC1qorPbBGBa2Gv4W0IISqynh6XmQRXU2SPQ
36 kat1dChaA8ykMs4BEwiqFYPiUBcbviaVkdFBUv6KHfrnWLC/IfuwaEm+EmIa+jAV
37 wmf2ntvYAbENpQzCnsxclhtgv4H7Jv//
38 -----END SESSION JOIN STRING-----
39
40 Into an interactive editor using:
41
42     rcodesign remote-sign --editor
43
44 Or into a new file whose path you define with:
45
46     rcodesign remote-sign --sjs-path /path/to/file/you/just/saved
47
48 (waiting for remote signer to join)
```

`rcodesign sign --remote-signer` prints out some instructions (including a wall of base64 encoded data) for what to do next. Importantly, it requests that someone else run `rcodesign remote-sign` to continue the signing process.

And here's a screenshot of me doing that from the Windows terminal:




```
connected to reader: Yubico YubiKey OTP+FIDO+CCID 0
using certificate in smartcard slot 9c
attempt 1/3
failed sign command with code 6982
device refused operation due to authentication error
Please enter device PIN: [hidden]
pin verification successful
attempt 2/3
connecting to wss://ws.codesign.gregoryszorc.com/
joining session...
successfully joined signing session 69bab200-128d-4f6e-a4d1-af609844910f
verifying encrypted communications with peer
waiting for server to send us a message...
waiting for server to send us a message...
creating signature for remote message: MYIB1DAYBgkqhkiG9w0BCQMxCwYJKoZIhvcNAQ
DYXJlbj9a17ujVhmFwvv/oTODAcBgkqhkiG9w0BCQUxDxcNMjIwNDI0MTg0OTE4WjCCASkGCSqGS
iBlbmNvZGluc29iVVRGLTgiPz4KPCFET0NUWVBFIHBsaXN0IFBVQkxJQyAiLS8vQXBwbGUvL0RUR
S5jb20vRFREcy9Qcm9wZXJ0eUxpc3QtMS4wLmR0ZCI+CjxwbGlzdCB2ZXJzaW9uPSIxLjAiPgo8Z
T4KCQk8ZGF0YT4KCQk3WTVQVllyeWVWRHhQRedtWUV3MkZ5Wlc0L1U9CgkJPJC9kYXRhPgoJPC9hc
AkCMS8wLQYJYIZIAWUDBAIBBCDtkj9VivJ5UPGMMaZgTDYXJlbj9a17ujVhmFwvv/oTOA==
initial signing attempt may fail if the certificate requires a pin to unlock
attempt 1/3
failed sign command with code 6982
device refused operation due to authentication error
Please enter device PIN: [hidden]
pin verification successful
attempt 2/3
sending signature to peer
relay acknowledged signature message received
```

This log shows us connecting and authenticating with the YubiKey along with some status updates regarding speaking to a remote server.

Finally, here's a screenshot of the GitHub Actions job output after I ran that command on my Windows machine:

```
48 (waiting for remote signer to join)
49 signer joined session; deriving shared encryption key
50 requesting signing certificate info from signer
51 remote signer will sign with certificate: Developer ID Application: Gregory Szorc (MK22MZF
52 registering signing key
53 automatically registered Apple CA certificate: Developer ID Certification Authority
54 automatically registered Apple CA certificate: Apple Root CA
55 using time-stamp protocol server http://timestamp.apple.com/ts01
56 automatically setting team ID from signing certificate: MK22MZF987
57 registering extra X.509 certificate
58 registering extra X.509 certificate
59 signing dist/input/rcodesign to dist/output/rcodesign
60 signing dist/input/rcodesign as a Mach-O binary
```

```
61 inferring default signing settings from Mach-O binary
62 preserving existing binary identifier in Mach-O
63 using team ID from settings
64 preserving code signature flags in existing Mach-O signature
65 setting binary identifier to rcodesign
66 parsing Mach-O
67 writing dist/output/rcodesign
68 signing Mach-O binary at index 0
69 attempting to derive code requirements from signing certificate
70 code requirements: 0: (identifier "rcodesign") and ((anchor apple generic) and ((certificate
    leaf[subject.OU] = "MK22MZP987"))));
71 binary targets macOS >= 11.0.0 with SDK 12.1.0
72 code directory version: 132096
73 creating cryptographic signature with certificate Developer ID Application: Gregory Szorc
74 Using time-stamp server http://timestamp.apple.com/ts01
75 sending signing request to remote signer
76 received signature from remote signer
77 total signature size: 186965 bytes
78 signing Mach-O binary at index 1
79 attempting to derive code requirements from signing certificate
80 code requirements: 0: (identifier "rcodesign-8d3eb894c5473a86") and ((anchor apple generic
    (certificate leaf[subject.OU] = "MK22MZP987"))));
81 binary targets macOS >= 11.0.0 with SDK 12.1.0
82 adding code signature flags from signing settings: ADHOC | LINKER_SIGNED
83 removing ad-hoc code signature flag
84 removing linker signed flag from code signature (we're not a linker)
85 code directory version: 132096
86 creating cryptographic signature with certificate Developer ID Application: Gregory Szorc
87 Using time-stamp server http://timestamp.apple.com/ts01
88 sending signing request to remote signer
89 received signature from remote signer
90 total signature size: 174199 bytes
91 terminating signing session on relay
92 relay server confirmed session termination
93 disconnecting from relay server
```

>  Upload

*Remote signing* enabled me to sign a macOS application from a GitHub Actions runner operated by GitHub while using a code signing certificate securely stored on my YubiKey plugged into a Windows machine hundreds of kilometers away from the GitHub Actions runner. Magic, right?

What's happening here is the 2 `rcodesign` processes are communicating with each other via websockets bridged by a central relay server. (I

operate a [default server free of charge](#). The server is open source and a Terraform module is available if you want to run your own server with hopefully just a few minutes of effort.) When the initiating machine wants to create a signature, it sends a message back to the *signer* requesting a cryptographic signature. The signature is then sent back to the initiator, who incorporates it.

**I designed this feature with automated releases from CI systems (like GitHub Actions) in mind. I wanted a way where I could streamline the code signing and release process of applications without having to give a low trust machine in CI ~unlimited access to my private signing key. But the more I thought about it the more I realized there are likely many other scenarios where this could be useful. Have you ever emailed or Dropboxed an application for someone else to sign because you don't have an Apple issued code signing certificate? Now you have an alternative solution that doesn't require copying files around!** As long as you can see the log output from the initiating machine or have that output communicated to you (say over a chat application or email), you can remotely sign files on another machine!

## An Aside on the Security of Remote Signing

At this point, I'm confident the more security conscious among you have been grimacing for a few paragraphs now. Websockets through a central server operated by a 3rd party?! Giving remote machines access to perform code signing against arbitrary content?! Your fears and skepticism are 100% justified: I'd be thinking the same thing!

I fully recognize that a service that facilitates remote code signing makes for a very lucrative attack target! If abused, it could be used to coerce parties with valid code signing certificates to sign unwanted code, like malware. There are many, many, many *wrong* ways to implement such a feature. I pondered for hours about the threat modeling and how to make

this feature as secure as possible.

[Remote Code Signing Design and Security Considerations](#) captures some of my high level design goals and security assessments. And [Remote Code Signing Protocol](#) goes into detail about the communications protocol, including the crypto (actual cryptography, not the fad) involved. The key takeaways are the protocol and server are designed such that a malicious server or man-in-the-middle can not forge signature requests. Signing sessions expire after a few minutes and 3rd parties (or the server) can't inject malicious messages that would result in unwanted signatures. There is an initial handshake to derive a session ephemeral shared encryption key and from there symmetric encryption keys are used so all meaningful messages between peers are end-to-end encrypted. About the worst a malicious server could do is conduct a denial of service. This is by design.

As I argue in [Security Analysis in the Bigger Picture](#), I believe that my implementation of *remote signing* is **more** secure than many common practices because common practices today entail making copies of private keys and giving low trust machines (like CI workers) access to private keys. Or files are copied around without cryptographic chain-of-custody to prove against tampering. Yes, *remote signing* introduces a vector for remote access to *use* signing keys. But practiced as I intended, *remote signing* can eliminate the need to copy private keys or grant ~unlimited access to them. From a threat modeling perspective, I think the net restriction in key access makes *remote signing* more secure than the private key management practices by many today.

**All that being said, the giant asterisk here is I implemented my own cryptosystem to achieve end-to-end message security. If there are bugs in the design or implementation, that cryptosystem could come crashing down, bringing defenses against message forgery with it.** At that point, a malicious server or privileged network actor could potentially

coerce someone into signing unwanted software. But this is likely the extent of the damage: an offline attack against the signing key should not be possible since signing requires presence and since the private key is never transmitted over the wire. Even without the end-to-end encryption, the system is *arguably* more secure than leaving your private key lingering around as an easily exfiltrated CI secret (or similar).

(I apologize to every cryptographer I worked with at Mozilla who beat into me the commandment that *thou shall not roll their own crypto*: I have sinned and I feel remorseful.)

Cryptography is hard. And I'm sure I made plenty of subtle mistakes. [Issue #552](#) tracks getting an audit of this protocol and code performed. And the aforementioned [protocol design docs](#) call out some of the places where I question decisions I've made.

**If you would be interested in doing a security review on this feature, please get in touch on issue #552 or [send me an email](#). If there's one immediate outcome I'd like from this blog post it would be for some white hat^Hknight to show up and give me peace of mind about the cryptosystem implementation.**

**Until then, please assume the end-to-end encryption is completely flawed.** Consider asking someone with *security* or *cryptographer* in their job title for their opinion on whether this feature is safe for you to use. Hopefully we'll get a security review done soon and this caveat can go away!

If you do want to use this feature, [Remote Code Signing](#) contains some usage documentation, including how to use it with GitHub Actions. (I could also use some help productionizing a reusable GitHub Action to make this more turnkey! Although I'm hesitant to do it before I know the cryptosystem is sound.)

That was a long introduction to *remote code signing*. But I couldn't in good faith present the feature without addressing the security aspect. Hopefully I didn't scare you away! **Traditional / local signing should have no security concerns** (beyond the willingness to run software written by somebody you probably don't know, of course).

## Apple Keychain Support

As of today's 0.14 release we now have early support for signing with code signing certificates stored in Apple Keychains! If you created your Apple code signing certificates in Keychain Access or Xcode, this is probably where your code signing certificates live.

I held off implementing this for the longest time because I didn't perceive there to be a benefit: if you are on macOS, just use Apple's official tooling. But with `rcodesign` gaining support for remote code signing and some other features that could make it a compelling replacement for Apple tooling on all platforms, I figured we should provide the feature so we stop discouraging people to export private keys from Keychains.

This integration is very young and there's still a lot that can be done, such as automatically using an appropriate signing certificate based on what you are signing. Please file feature request issues if there's a must-have feature you are missing!

## Better Debugging of Failures

Apple's code signing is complex. It is easy for there to be subtle differences between Apple's tooling and `rcodesign`.

`rcodesign` now has `print-signature-info` and `diff-signatures` commands to dump and compare YAML metadata pertinent to code signing to make it easier to compare behavior between code signing implementations and even multiple signing operations.



The documentation around [debugging and reporting bugs](#) now emphasizes using these tools to help identify bugs.

## A Request For Users and Feedback

I now believe `rcodesign` to be generally usable. I've thrown a lot of random software at it and I feel like most of the big bugs and major missing features are behind us.

But I also feel it hasn't yet received wide enough attention to have confidence in that assessment.

**If you want to help the development of this tool, the most important actions you can take are to attempt signing / notarization operations with it and report your results.**

Does `rcodesign` spark joy? Please leave a comment in the [GitHub discussion for the latest release](#)!

Does `rcodesign` not work? I would very much appreciate a bug report! Details on how to file good bugs are [in the docs](#).

Have general feedback? UI is confusing? Documentation is insufficient? Leave a comment in the aforementioned discussion. Or [create a GitHub issue](#) if you think it is actionable. I can't fix what I don't know about!

Have private feedback? [Send me an email](#).

## Conclusion

I could write thousands of words about all I learned from hacking on this project.

I've learned way too much about too many standards and specifications in the crypto space. RFCs 2986, 3161, 3280, 3281, 3447, 4210, 4519, 5280,



5480, 5652, 5869, 5915, 5958, and 8017 plus probably a few more. How cryptographic primitives are stored and expressed: ASN.1, OIDs, BER, DER, PEM, SPKI, PKCS#1, PKCS#8. You can show me the raw parse tree for an ASN.1 data structure and I can probably tell you what RFC defines it. I'm not proud of this. But I will say actually knowing what every field in an X.509 certificate does or the many formats that cryptographic keys are expressed in seems empowering. Before, I would just search for the `openssl` incantation to do something. Now, I know which ASN.1 data structures are involved and how to manipulate the fields within.

I've learned way too much around minutia around how Apple code signing actually works. The mechanism is way too complex for something in the security space. There was at least one high profile Gatekeeper bug in the past year allowing improperly signed code to run. I suspect there will be more: the surface area to exploit is just too large.

**I think I'm proud of building an open source implementation of Apple's code signing. To my knowledge nobody else has done this outside of Apple. At least not to the degree I have.** Then factor in that I was able to do this without access (or willingness) to look at Apple source code and much of the progress was achieved by diffing and comparing results with Apple's tooling. Hours of staring at diffoscope and comparing binary data structures. Hours of trying to find the magical settings that enabled a SHA-1 or SHA-256 digest to agree. It was tedious work for sure. I'll likely never see a financial return on the time equivalent it took me to develop this software. But, I suppose I can nerd brag that I was able to implement this!

**But the real reward for this work will be if it opens up avenues to more (open source) projects distributing to the Apple ecosystems.** This has historically been challenging for multiple reasons and many open source projects have avoided official / proper distribution channels to avoid the pain (or in some cases because of philosophical disagreements with the

premise of having a walled software garden in the first place). I suspect things will only get worse, as I feel it is inevitable Apple clamps down on signing and notarization requirements on macOS due to the rising costs of malware and ransomware. **So having an alternative, open source, and multi-platform implementation of Apple code signing seems like something important that should exist in order to provide opportunities to otherwise excluded developers. I would be humbled if my work empowers others. And this is all the reward I need.**