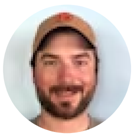
Good certificates die young: what's passive revocation and how is it implemented?

Updated on: May 20, 2024

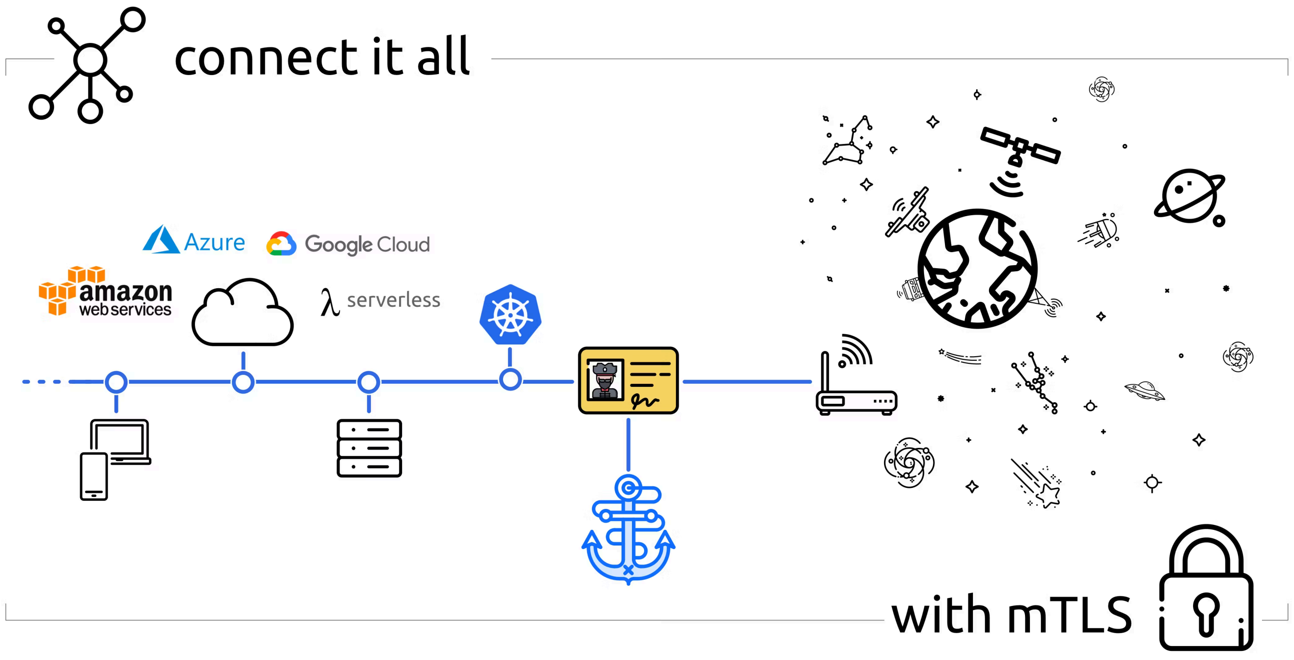


Mike Malone

Follow Smallstep

If you're a normal human person you probably don't think much about certificate revocation. This post will help you justify your apathy. It will explain why your indifference is, in fact, the technically correct attitude to have regarding this particular detail of your system's security architecture.

But first, let's talk certificates more generally. Certificates deserve more attention. They're underappreciated and underutilized. Over the past few years HTTPS adoption on the web has sky-rocketed (largely thanks to Let's Encrypt). It's time for our internal systems to get the same treatment: communication between microservices, containers, functions, connected devices, and whatever else constitutes "production" should be at least as secure as our day-to-day web browsing.



Luckily, TLS — the protocol behind HTTPS — is not limited to web browsers and servers. TLS is ubiquitous. There are implementations in every standard language library. It's supported by proxies, queues, databases, and pretty much everything else. So, you can use mutual TLS (mTLS) to securely connect stuff, in any language, whether that stuff is on-prem or in the cloud, running across multiple clouds, at the edge, or anywhere else. All without complex network-level magic. If you need to securely connect stuff, you should use TLS.

To do so, all you need are certificates.

Turns out getting certificates to all the places they're needed is easier said than done. Harder still is automating these processes without compromising security. We built `step` and `step certificates` to make automated certificate management easy.

☆ Star smallstep/cli 3,803 ☆ Star smallstep/certificates 7,126

`step` lets you spin up your own certificate authority (CA) and issue, renew, and (as of today) revoke certificates.

But I digress. Let's back up. What even is a certificate, and *why* are they so useful?

Mutual TLS uses public/private key pairs (a.k.a. asymmetric key pairs) for authentication. The magic of asymmetric cryptography is that we can *sign* some data using a private key and someone else can *verify* our signature using the corresponding public key.

Critically, you can **only verify** a signature using a public key. You **can't generate** one. You need the private key to do that.

If you know my public key, you can authenticate my identity via the following protocol:

1. You generate a big random number and send it to me
2. I sign your big random number with my private key and send you my signature
3. You verify my signature using my public key

If I'm the only one that knows my private key, then I'm the only one who could have generated the signature. So, you must be talking to me. That's roughly how TLS works. With mutual TLS I'd also challenge you to prove your identity to me.

For this to work, you need to know my public key. What if you don't? That's why we have certificates. **A certificate binds a name to a public key.** They're issued by a *certificate authority* (CA) (i.e., they're *signed* by the CA's private key). Before running the authentication protocol described above, I can simply send you my certificate, which you can verify and decode to extract my public key.

Certificates elegantly solve the public key distribution problem. Instead of knowing the public key of everyone you want to talk to, you only need to know their names and the public key of a CA you trust to issue certificates.

This sort of infrastructure — CAs and protocols and such for managing and distributing public keys — is called *public key infrastructure* (PKI). The beautiful thing about certificate-based PKI is that, once certificates are issued, it's completely decentralized. I can send you my certificate and you can validate it without runtime assistance from any central authority. Thus, certificate-based PKI is inherently fault tolerant and trivial to scale. This is a good thing.

But decentralization has a down side: there's no way to actively disseminate certificate revocation information to the *relying parties* that use certificates. In other words, once a certificate is issued, you can't un-issue it. Certificates do eventually expire but, until then, a compromised private key can be used to impersonate the certificate owner. This is not a good thing.

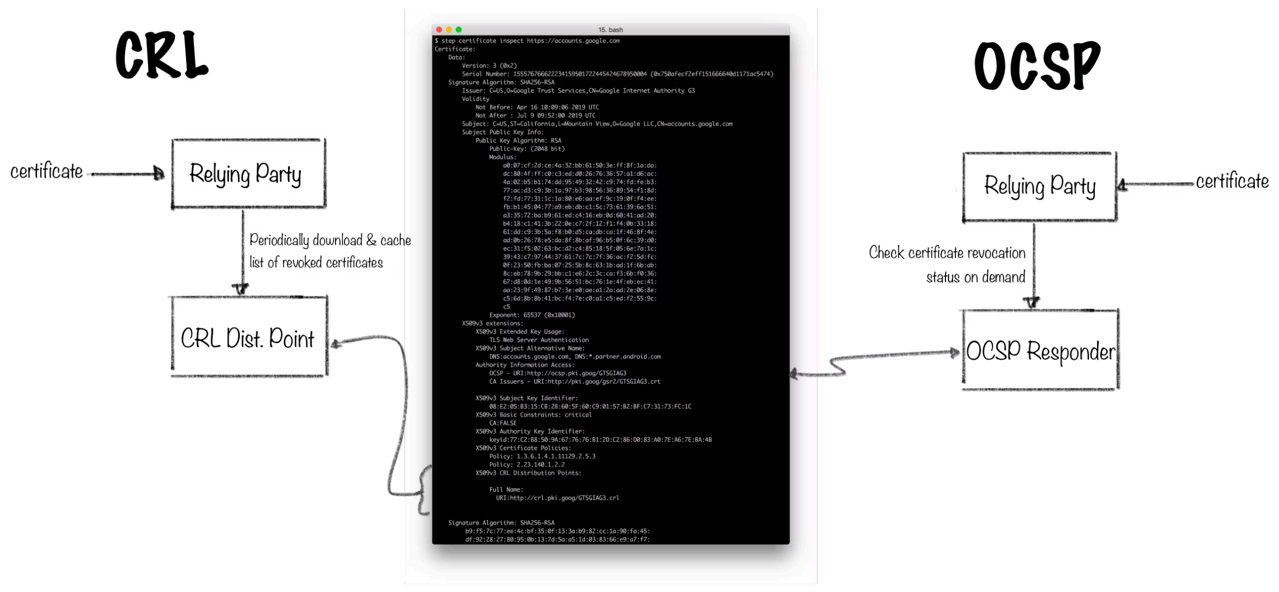
Revocation

At a glance this seems like a straightforward enough problem. The obvious solution is to simply build a mechanism for *revoking* a certificate: immediately marking a certificate as invalid despite its genuine appearance.

Thankfully, we needn't invent such a mechanism. Two standard mechanisms for certificate revocation already exist:

1. Certificate Revocation Lists (CRLs) list the serial numbers of revoked certificates in one big downloadable data structure, and
2. Online Certificate Status Protocol (OCSP) defines an API for determining whether a particular certificate has been revoked.

Unfortunately, both of these mechanisms have pretty major issues.


CRL
certificate
Relying Party
Periodically download & cache list of revoked certificates
CRL Dist. Point
15. bash
OCSP
Relying Party
certificate
Check certificate revocation status on demand
OCSP Responder

```
$ step certificate inspect https://accounts.google.com
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 155576766622234159501722445424678950004 (0x750afecf2eff151666640d1171ac5474)
    Signature Algorithm: SHA256-RSA
        Issuer: C=US,O=Google Trust Services,CN=Google Internet Authority G3
        Validity
            Not Before: Apr 16 10:09:06 2019 UTC
            Not After : Jul 9 09:52:00 2019 UTC
        Subject: C=US,ST=California,L=Mountain View,O=Google LLC,CN=accounts.google.com
        Subject Public Key Info:
            Public Key Algorithm: RSA
                Public-Key: (2048 bit)
                Modulus:
                    a0:07:cf:2d:ce:4a:32:bb:61:50:3e:ff:8f:1a:da:
                    dc:80:4f:ff:c0:c3:ed:d0:26:76:36:57:a1:d6:ac:
                    4a:02:b5:b1:74:dd:95:49:32:42:c9:74:fd:fe:b3:
                    77:ac:d3:c9:3b:1a:97:b3:98:56:36:89:54:f1:8d:
                    f2:fd:77:31:1c:1a:80:e6:aa:ef:9c:19:0f:f4:ee:
                    fb:b1:45:04:77:a9:eb:db:c1:5c:73:61:39:6a:51:
                    a3:35:72:ba:b9:61:ed:c4:16:eb:0d:60:41:ad:20:
                    b4:18:c1:41:3b:22:0e:c7:2f:12:f1:f4:0b:33:18:
                    61:dd:c9:3b:5a:f8:b0:d5:ca:db:ca:1f:46:8f:4e:
                    ad:0b:26:78:e5:da:8f:8b:af:96:b5:0f:6c:39:d0:
                    ec:31:f5:02:63:bc:d2:c4:85:18:5f:05:6e:7a:1c:
                    39:43:c7:97:44:37:61:7c:7c:7f:36:ac:f2:5d:fc:
                    0f:23:50:fb:ba:07:25:5b:8c:63:1b:ad:1f:6b:ab:
                    8c:eb:78:9b:29:bb:c1:e6:2c:3c:ca:f3:6b:f0:36:
                    67:d8:0d:1e:49:9b:56:51:bc:76:1e:4f:eb:ec:41:
                    aa:23:9f:49:87:b7:3e:e0:ae:a1:2a:ad:2e:06:8e:
                    c5:6d:8b:8b:41:bc:f4:7e:c0:a1:c5:ed:f2:55:9c:
                    c5
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Subject Alternative Name:
                DNS:accounts.google.com, DNS:*.partner.android.com
            Authority Information Access:
                OCSP - URI:http://ocsp.pki.goog/GTSGIAG3
                CA Issuers - URI:http://pki.goog/gsr2/GTSGIAG3.crt

            X509v3 Subject Key Identifier:
                08:E2:05:B3:15:CB:2B:60:5F:60:C9:01:57:B2:BF:C7:31:73:FC:1C
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Authority Key Identifier:
                keyid:77:C2:B8:50:9A:67:76:76:B1:2D:C2:86:D0:83:A0:7E:A6:7E:BA:4B
            X509v3 Certificate Policies:
                Policy: 1.3.6.1.4.1.11129.2.5.3
                Policy: 2.23.140.1.2.2
            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://crl.pki.goog/GTSGIAG3.crl


    Signature Algorithm: SHA256-RSA
         b9:f5:7c:77:ee:4c:bf:35:0f:13:3a:b9:82:cc:1a:90:fa:45:
         df:92:28:27:80:95:0b:13:7d:5a:a5:1d:03:83:66:e9:a7:f7:
```

CRLs generally contain the serial number of *every* revoked certificate. In a large dynamic system, with services and machines coming and going continuously, this list will get big fast. Downloading a huge list of revoked certificates every time a connection is established would be slow and resource-intensive, so CRLs are usually cached. Caching creates a delay between when a certificate is revoked and when the rest of your system notices. Thus, revocation is not immediate.

OCSP responses may also be cached, with the same consequence. However, caching is less necessary because OCSP responses are relatively small. But, without caching, you get a different problem. Checking revocation status using OCSP requires a blocking request to an *OCSP responder* every time a certificate is validated (i.e., every time a connection is established). Obviously, this adds latency.

CRL and OCSP also add significant architectural and operational complexity: they both take a beautifully distributed authentication system and add a central choke point that could become a scale bottleneck or, worse, take your entire system down. In theory it seems like highly available and scalable CRL & OCSP infrastructure should exist. Unfortunately, as far as I know, it doesn't. There's no off-the-shelf open source solution here. (This is something we're working on at smallstep.)

Even if good CRL & OCSP infrastructure did exist, there's a much hairier problem with these protocols: lack of support amongst relying party implementations. While the core TLS protocol is supported almost everywhere, CRL and OCSP are not. By default, most TLS implementations (outside of web browsers) don't check CRL or OCSP revocation status. In other words, it's very likely that your favorite language's TLS implementation will happily accept a revoked certificate.

The non-trivial operational challenge of building and operating highly available CRL & OCSP infrastructure, combined with the lack of good support for these protocols by PKI participants, makes reliable active revocation extremely challenging to deliver in practice.

So where does this leave us? Is there a better way?

Passive Revocation

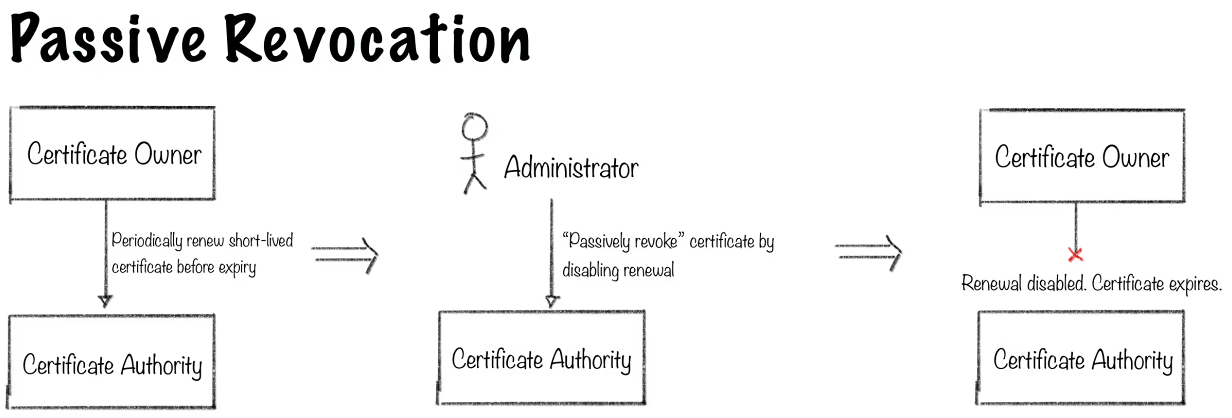
We want the ability to quickly invalidate the binding between a name and a public key globally, without centralized infrastructure, and without relying on any special features of TLS that might not be supported everywhere.

The answer is simple: *short-lived certificates*.

Again, certificates expire. After expiration, they're no longer trusted. This happens to be a core characteristic that every non-broken TLS implementation will implement correctly.

Historically, without automation, certificates were issued with fairly long lifetimes: months or years. Shorter lifetimes simply weren't feasible. Frequent manual renewals would be a full-time job. This is no longer a problem. Operational automation allows us to make certificate lifetimes *much* shorter: days, hours, even minutes.

If we make certificate lifetimes short enough, we can leverage expiration to approximate revocation: instead of *killing* a certificate by actively revoking it before it expires, we can disable renewal for a certificate and simply *let it die*. This is called *passive revocation* (which is really just a fancy way of saying let's not explicitly do revocation at all).


Passive Revocation
Certificate Owner
Periodically renew short-lived certificate before expiry
Certificate Authority
Administrator
"Passively revoke" certificate by disabling renewal
Certificate Authority
Certificate Owner
Renewal disabled. Certificate expires.
Certificate Authority

Admittedly, passive revocation does leave a window of time during which an attacker might be able to misuse a certificate. But even if you absolutely must have immediate revocations, you'll still want passive revocation and short-lived certificates for two reasons:

1. The vast majority of certificates fall out of use for administrative reasons: no key compromise has occurred; the certificate is simply no longer needed. Passive revocation can be used here instead of burdening your CRL or OCSP infrastructure.
2. Certificates can be culled from CRLs and/or from your OCSP responder's active set once they've expired. So keeping certificate lifetimes short will make these other mechanisms more efficient.

So, you're going to want passive revocation, even if you also implement some other revocation mechanism.

Let's walk through what's required to build an internal PKI with passive revocation, using `step` to demonstrate. Ultimately what we need are: 1) short-lived certificates, with 2) automated renewal, and 3) some way to disable renewal for "passively revoked" certificates.

Install step (`brew install step`) to follow along at home.

SHORT-LIVED CERTIFICATES

First, let's initialize a new PKI and start the step CA. We'll write a password out to `password.txt` so we don't have to enter it repeatedly.

```
$ echo "p4ss" > password.txt

$ step ca init --name "Local CA" --provisioner admin \
               --dns localhost --address ":443" \
               --password-file password.txt \
               --provisioner-password-file password.txt

$ step-ca $(step path)/config/ca.json --password-file password.txt
```

Now let's generate a single-use bootstrap token and use it to obtain a certificate. (In a real production infrastructure something in your software delivery pipeline — puppet, chef, ansible, kubernetes — would generate this token and hand it to whatever needs a certificate):

```
provisioner$ step ca token --password-file password.txt foo.local
✔ Key ID: w1OUFng_fCqWygHHpc9Ak8m_HGmE0TEasYIfahLoZUg (admin)
eyJhbGciOiJFUzI1NiIsImtpZCI6IncxT1VGbmdfZkNxV3lnSEhwYzlBazhtX0hHbUUwVEVh
```

Now we can generate a keypair and use our token to obtain a certificate for `foo.local` from our CA:

```
foo$ step ca certificate foo.local foo.crt foo.key --token eyJhb...
✔ CA: https://localhost
✔ Certificate: foo.crt
✔ Private Key: foo.key

foo$ step certificate inspect --short foo.crt
X.509v3 TLS Certificate (ECDSA P-256) [Serial: 2599...1204]
  Subject:     foo.local
  Issuer:      Local CA Intermediate CA
  Provisioner: admin [ID: w1OU...oZUg]
  Valid from:  2019-05-01T21:06:25Z
          to:  2019-05-02T21:06:25Z
```

As demonstrated here, the default lifetime of certificates issued by `step-ca` is 24 hours. There's no technical definition of "short-lived", but this is suitably short for many scenarios. If it's not right for you, you can adjust the `defaultTLSCertDuration` per-provisioner or pass the `--not-after` flag to `step ca certificate` to adjust the lifetime of an individual certificate. Shorter lifetimes (e.g., five minutes or so) are better from a security perspective, but we'll soon find there's some operational pressure pushing in the other direction.

AUTOMATED RENEWAL

With short certificate lifetimes, our services and devices will likely out-live their certificates. We need to renew certificates that are still in use, extending their lifetimes before they expire. Again, `step` makes this very easy:

```
foo$ step ca renew --force foo.crt foo.key
Your certificate has been saved in foo.crt.

foo$ step certificate inspect --short foo.crt
X.509v3 TLS Certificate (ECDSA P-256) [Serial: 1664...3445]
  Subject:     foo.local
  Issuer:      Local CA Intermediate CA
  Provisioner: admin [ID: w1OU...oZUg]
```

```
      Valid from:  2019-05-01T21:15:16Z
              to:  2019-05-02T21:15:16Z
```

Note the change in the validity period relative to the original certificate above.

Renewing a certificate and updating a file in-place on disk is a great start. But a process that uses a certificate will typically only read it from disk once, when it starts up. Luckily, it's common for infrastructure like `nginx` to respond to a `SIGHUP` by reloading configuration files and other artifacts, including certificates.

Rather than renewing immediately and exiting, we can pass a couple flags to tell `step ca renew` to daemonize, renew the certificate periodically, and execute a command to `HUP` our hypothetical `nginx` proxy after each renewal:

```
foo$ step ca renew --daemon --exec "kill -HUP $NGINX_PID" foo.crt foo.ke
INFO: 2019/05/01 14:22:18 first renewal in 15h50m43s
```

This completely automates renewals for this scenario.

If you're terminating TLS in your apps instead of using a proxy, good for you! But you will have to write a bit of additional logic to handle renewals. To help, we've started collecting production-grade examples of these procedures in various programming languages in a sub-repository of step certificates called hello mTLS. We're just getting started but we've got a few good examples there already and are happily awaiting contributions!

Once daemonized, the `renew` command waits until the certificate's lifetime is two-thirds elapsed before attempting a renewal (e.g., it waits 16 hours then starts renewing 8 hours before a certificate with a 24 hours lifetime expires). Renewals are retried if the CA is unavailable, so there's some operational leeway if the CA goes down.

This is the operational pressure towards longer-lived certificates that we alluded to earlier. If certificates live for 24 hours our CA can be down for 8 hours before certificates start expiring. That's a decent window in which to receive an alert and remediate. But if we push certificate lifetimes down to a couple minutes, we'll need to make sure our CA doesn't go down for more than a minute or two. That's harder. Furthermore, since certificates are issued and validated on different servers, very short certificate lifetimes require precise clock synchronization across your entire infrastructure. All of this is technically possible but requires a lot of operational discipline to pull off. Know thyself and beware.

PASSIVELY REVOKING A CERTIFICATE

Renewal allows a certificate owner to extend the lifetime of a certificate before it expires. Unfortunately, it also lets an attacker with the right private key do the same thing. To prevent this, we need some way to tell the certificate authority not to renew a particular certificate. The most recent release of `step` (`v0.10.0`) adds this functionality.

If a private key is compromised, we can use `step` to *passively revoke* a certificate, disabling renewals:

```
foo$ step ca revoke --cert foo.crt --key foo.key
Certificate with Serial Number 48122447568876458238057740509229876261 ha
```

Any subsequent attempt to renew a passively revoked certificate will fail,

```
foo$ step ca renew foo.crt foo.key
error renewing certificate: Unauthorized
```

and the attempt will be reported in the CA logs:

```
WARN[0841] duration="103.709µs" duration-ns=103709 error="renew: certifi
```

This may seem like a small enhancement (and it is). But it's important. Passive revocation is the right way to handle certificate revocation 99% of the time, and this addition to `step` and `step-ca` makes passive revocation dead simple.

So you were right all along

Certificate-based PKI is awesome. But you do need to have a plan in place to handle a private key compromise. At first this seems like an easy enough problem to solve, but it's fraught. CRL and OCSP are operationally complex and poorly supported by relying parties. A shoddy implementation can compromise the scalability, efficiency, and availability of your entire system.

Turns out this is one of those happy situations where the right answer is also the easy one: don't do revocations at all. Instead, issue short-lived certificates. Automate renewals, and use *passive revocation* to disable renewal for certificates with compromised private keys. Then simply let them die. This strategy is easier to implement, harder to misuse, and the resulting system is easier to operate. If you need help, check out `step` :

☆ Star smallstep/cli 3,803 ☆ Star smallstep/certificates 7,126

All you normal human people were (kinda) right all along: if you set things up properly, certificate revocation isn't worth worrying about. But certificates are. If you don't already have an internal PKI you should get yourself one. And start using TLS everywhere. It's the right thing to do.

Mike Malone has been working on making infrastructure security easy with Smallstep for six years as CEO and Founder. Prior to Smallstep, Mike was CTO at Betable. He is at heart a distributed systems enthusiast, making open source solutions that solve big problems in Production Identity and a published research author in the world of cybersecurity policy.

Step Certificates Technical

Further Reading