Private Certificate Creation Using OpenSSL:

A Complete Tutorial

Public certificates are the right tool when the whole internet needs to trust your identity. But plenty of situations call for the opposite: internal tools, development environments, and closed systems where a private certificate, one issued and trusted only within your own organization, is exactly what is needed. OpenSSL, the venerable open-source cryptography toolkit, remains the standard way to build this kind of private certificate infrastructure from scratch. This tutorial walks through the full process.

When Private Certificates Make Sense

Private certificates are appropriate whenever the relying parties are known and controlled by you: internal APIs, service-to-service communication inside a private network, development and staging environments, VPN client authentication, and increasingly, authentication between internal microservices and the AI agents that call them. Because a private certificate is not vouched for by any public root trusted by browsers and operating systems by default, it requires you to distribute your own root certificate to every system that needs to trust it, which is entirely manageable inside an organization you control.

Prerequisites

This tutorial assumes OpenSSL is already installed, which is the case by default on most Linux distributions and macOS systems, and available for Windows through several distributions. It also assumes a basic comfort with the command line, since every step below is performed through OpenSSL's command-line interface.

01

Create the Root Certificate Authority Key

The root CA key is the single most important piece of key material in the entire private PKI, since anyone possessing it can issue certificates that everything downstream will trust. Generate it with a strong key size, for example: openssl genrsa -aes256 -out rootCA.key 4096. The -aes256 flag encrypts the key file with a passphrase, which is strongly recommended for a root key given how much trust rests on it.

02

Create the Root Certificate

With the root key generated, create a self-signed root certificate: openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 3650 -out rootCA.pem. This command will prompt for identifying details such as organization name and common name. The resulting rootCA.pem is the file that needs to be distributed to and trusted by every system that will rely on certificates issued from this private CA.

03

Create an Intermediate CA

Following the same two-tier structure used by public CAs, it is good practice to generate an intermediate key and certificate signed by the root, then use the intermediate for day-to-day issuance rather than exposing the root key to routine operations. This mirrors the exact reasoning covered elsewhere in this series: keeping the root offline and rarely used dramatically reduces the risk of catastrophic compromise.

04

Generate an End-Entity Key and CSR

For each server or service that needs a certificate, generate its own key pair and Certificate Signing Request: openssl genrsa -out server.key 2048, followed by openssl req -new -key server.key -out server.csr. Include the correct Subject Alternative Names for every hostname the certificate needs to cover; modern clients generally ignore the legacy Common Name field and check Subject Alternative Names exclusively.

05

Sign the Certificate With Your CA

Using the intermediate (or root, for smaller setups) key and certificate, sign the CSR to produce the final certificate: openssl x509 -req -in server.csr -CA intermediateCA.pem -CAkey intermediateCA.key -CAcreateserial -out server.crt -days 397 -sha256 -extfile server.ext, where server.ext contains the Subject Alternative Name extension and any other required extensions. The resulting server.crt, along with the intermediate certificate, forms the chain that gets installed on the actual server.

06

Trust the Private Root

For any of this to work, every client that needs to connect to a service using these certificates must have the root certificate, rootCA.pem, added to its trusted certificate store. This can be done manually for a handful of machines or pushed out via configuration management tooling across a fleet, and is a step frequently overlooked by beginners, leading to confusing trust errors even though the certificate itself was generated correctly.

Using This for AI-Facing Internal Services

A private CA built this way is a natural fit for issuing certificates to internal AI model-serving endpoints and the microservices that surround them, since these are almost always internal-only, high-volume, and benefit from short lifespans and heavy automation rather than the manual process described above. In practice, most organizations running AI infrastructure at any scale wrap this OpenSSL workflow inside automated tooling, such as a step-ca instance or an internal ACME server, so that AI services and the pipelines that deploy them can request and rotate certificates without a human running these commands by hand each time. Understanding the manual OpenSSL process first, however, makes it far easier to reason about what that automation is actually doing under the hood.

Final Notes

Building a private PKI with OpenSSL gives an organization complete control over its internal trust infrastructure, at the cost of taking on the full responsibility for protecting the root key and distributing trust correctly. For small, well-understood environments this manual approach is often sufficient. For anything approaching production scale, particularly environments serving AI workloads with high certificate turnover, this manual foundation should be automated as quickly as possible, a topic covered in more depth in the companion article on building a full internal Certificate Authority.

The Countdown Is Already Running: 200 Days, 100 Days, 47 Days

Every certificate conversation in 2026 eventually arrives at the same clock, and it is worth closing on it here. The CA/Browser Forum's Ballot SC-081v3 is not a proposal under discussion; it is an approved, already-in-motion schedule. Maximum public TLS certificate lifetimes fall from 398 days to 200 days on March 15, 2026. They fall again to 100 days on March 15, 2027. By March 15, 2029, they drop to just 47 days, with domain validation itself needing to be re-proven roughly every 10 days.

01

Translate that into operational terms and the picture gets stark quickly. An organization currently renewing certificates a few times a year will be handling renewal events on the order of every couple of weeks by the end of this countdown, across every endpoint it operates. Manual tracking, calendar reminders, and a spreadsheet somebody checks once a month will not survive contact with that cadence. What has always been an occasional chore is becoming a continuous, automated operation, whether an organization plans for it or not.

02

Private PKI built with OpenSSL is technically exempt from the public lifetime rules below, but most organizations run a hybrid estate, and the automation discipline the public side is being forced into tends to pull private certificate practices toward the same short-lived, automated model.

03

The 200-day, 100-day, and 47-day milestones are not distant hypotheticals; the first has already arrived. Organizations that build the automation loop now, generating keys, vaulting them securely, brokering issuance across Certificate Authorities through APIs, and rebinding certificates to live endpoints without manual intervention, will meet each deadline without disruption. Organizations that wait will be rebuilding their certificate operations under deadline pressure, with far less room for error and far less time to get it right. The countdown is the call to action. The only real decision left is whether to automate on your own schedule, or on the CA/Browser Forum's.