CHAPTER 11

Case Study: Designing, Implementing, and Maintaining a Publicly Trusted CA

By Andy Warner, James Kasten, Rob Smits, Piotr Kucharski, and Sergey Simakov

SREs, developers, and administrators sometimes need to interact with certificate authorities (CAs) to obtain certificates for encryption and authentication, or to handle functionality like VPNs and code signing. This case study focuses on Google's journey of designing, implementing, and maintaining a publicly trusted CA using the best practices discussed in this book. This journey was informed by internal scalability needs, compliance requirements, and security and reliability requirements.

Background on Publicly Trusted Certificate Authorities

Publicly trusted certificate authorities act as trust anchors for the transport layer of the internet by issuing certificates for Transport Layer Security (TLS),[1] S/MIME,[2] and other common distributed trust scenarios. They are the set of CAs that browsers, operating systems, and devices trust by default. As such, writing and maintaining a publicly trusted CA raises a number of security and reliability considerations.

To become publicly trusted and maintain that status, a CA must pass a number of criteria that span different platforms and use cases. At minimum, publicly trusted CAs must undergo audits against standards such as WebTrust and those set by organizations like the European Telecommunications Standards Institute (ETSI).

1 The latest version of TLS is described in RFC 8446.

2 Secure/Multipurpose Internet Mail Extensions is a common method to encrypt email content.

Publicly trusted CAs must also meet the objectives of the CA/Browser Forum Baseline Requirements. These evaluations assess logical and physical security controls, procedures, and practices, and a typical publicly trusted CA spends at least one quarter of each year on these audit(s). Additionally, most browsers and operating systems have their own unique requirements that a CA must meet before it's trusted by default. As requirements change, CAs need to be adaptable and amenable to making process or infrastructure changes.

Chances are, your organization will never need to build a publicly trusted CA[3]—most organizations rely on third parties for acquiring public TLS certificates, code signing certificates, and other types of certificates that require broad trust by users. With that in mind, the goal of this case study is not to show you how to build a publicly trusted CA, but to highlight some of our findings that might resonate with projects in your environment. Key takeaways included the following:

- Our choice of programming language and decision to use segmentation or containers when handling data generated by third parties made the overall environment more secure.
- Rigorously testing and hardening code—both code we generated ourselves and third-party code—was critical for addressing fundamental reliability and security issues.
- Our infrastructure became safer and more reliable when we reduced complexity in the design and replaced manual steps with automation.
- Understanding our threat model enabled us to build validation and recovery mechanisms that allow us to better prepare for a disaster ahead of time.

Why Did We Need a Publicly Trusted CA?

Our business needs for a publicly trusted CA changed over time. In Google's early days, we purchased all of our public certificates from a third-party CA. This approach had three inherent problems we wanted to solve:

Reliance on third parties

Business requirements that necessitate a high level of trust—for example, offering cloud services to customers—meant we needed strong validation and control over how certificates were issued and handled. Even if we performed mandatory audits in the CA ecosystem, we were unsure of whether third parties could meet

3 We recognize that many organizations do build and operate private CAs, using common solutions such as Microsoft's AD Certificate Services. These are typically for internal use only.

a high standard of safety. Notable lapses in security at publicly trusted CAs solidified our views about safety.[4]

Need for automation
: Google has thousands of company-owned domains that serve users globally. As part of our ubiquitous TLS efforts (see "Example: Increasing HTTPS usage" on page 136), we wanted to protect every domain we own and rotate certificates frequently. We also wanted to provide an easy way for customers to get TLS certificates. Automating the acquisition of new certificates was difficult because third-party publicly trusted CAs often did not have extensible APIs, or provided SLAs below our needs. As a result, much of the request process for these certificates involved error-prone manual methods.

Cost
: Given the millions of TLS certificates Google wanted to use for its own web properties and on behalf of customers, cost analysis showed it would be more cost-effective to design, implement, and maintain our own CA rather than continuing to obtain certificates from third-party root CAs.

The Build or Buy Decision

Once Google decided it wanted to operate a publicly trusted CA, we had to decide whether to buy commercial software to operate the CA or to write our own software. Ultimately, we decided to develop the core of the CA ourselves, with the option to integrate open source and commercial solutions where necessary. Among a number of deciding factors, there were a few primary motivators behind this decision:

Transparency and validation
: Commercial solutions for CAs often didn't come with the level of auditability for code or the supply chain that we needed for such critical infrastructure. Even though it was integrated with open source libraries and used some third-party proprietary code, writing and testing our own CA software gave us increased confidence in the system we were building.

Integration capabilities
: We wanted to simplify implementation and maintenance of the CA by integrating with Google's secure critical infrastructure. For example, we could set up regular backups in Spanner with one line in a configuration file.

[4] DigiNotar went out of business after attackers compromised and misused its CA.

Flexibility

The wider internet community was developing new initiatives that would provide increased security for the ecosystem. Certificate Transparency—a way to monitor and audit certificates—and domain validation using DNS, HTTP, and other methods[5] are two canonical examples. We wanted to be early adopters of these kinds of initiatives, and a custom CA was our best option for being able to add this flexibility quickly.

Design, Implementation, and Maintenance Considerations

To secure our CA, we created a three-layer tiered architecture, where each layer is responsible for a different part of the issuance process: certificate request parsing, Registration Authority functions (routing and logic), and certificate signing. Each layer is composed of microservices with well-defined responsibilities. We also devised a dual trust zone architecture, where untrusted input is handled in a different environment than critical operations. This segmentation creates carefully defined boundaries that promote understandability and ease of review. The architecture also makes mounting an attack more difficult: since components are limited in functionality, an attacker who gains access to a given component will be similarly limited in the functionality they can affect. To gain additional access, the attacker would have to bypass additional audit points.

Each microservice is designed and implemented with simplicity as a key principle. Over the lifetime of the CA, we continually refactor each component with simplicity in mind. We subject code (both internally developed and third-party) and data to rigorous testing and validation. We also containerize code when doing so will improve safety. This section describes our approach to addressing security and reliability through good design and implementation choices in more detail.

5 A good reference for domain validation guidelines is the CA/Browser Forum Baseline Requirements.

Evolving Design and Implementation

This case study presents a somewhat idealized picture of good system design, but in reality the design choices we discuss were made over the course of nearly a decade, in three distinct design iterations. New requirements and practices emerged over a long period of time. It's often not possible to design and implement every aspect of security and reliability from the start of a project. However, establishing good principles from the outset is important. You can start small, and continuously iterate on the security and reliability properties of the system to achieve long-term sustainability.

Programming Language Choice

The choice of programming language for parts of the system that accept arbitrary untrusted input was an important aspect of the design. Ultimately, we decided to write the CA in a mix of Go and C++, and chose which language to use for each subcomponent based upon its purpose. Both Go and C++ have interoperability with well-tested cryptographic libraries, exhibit excellent performance, and have a strong ecosystem of frameworks and tools to implement common tasks.

Since Go is memory-safe, it has some additional upsides for security where the CA handles arbitrary input. For example, Certificate Signing Requests (CSRs) represent untrusted input into the CA. CSRs could come from one of our internal systems, which may be relatively safe, or from an internet user (perhaps even a malicious actor). There is a long history of memory-related vulnerabilities in code that parses DER (Distinguished Encoding Rules, the encoding format used for certificates),[6] so we wanted to use a memory-safe language that provided extra security. Go fit the bill.

C++ is not memory-safe, but has good interoperability for critical subcomponents of the system—especially for certain components of Google's core infrastructure. To secure this code, we run it in a secure zone and validate all data before it reaches that zone. For example, for CSR handling, we parse the request in Go before relaying it to the C++ subsystem for the same operation, and then compare the results. If there is a discrepancy, processing does not proceed.

Additionally, we enforce good security practices and readability at pre-submit time for all C++ code, and Google's centralized toolchain enables various compile-time and runtime mitigations. These include the following:

6 The Mitre CVE database contains hundreds of vulnerabilities discovered in various DER handlers.

W^X
: Breaks the common exploitation trick of `mmap`ing with `PROT_EXEC` by copying shellcode and jumping into that memory. This mitigation does not incur a CPU or memory performance hit.

Scudo Allocator
: A user-mode secure heap allocator.

SafeStack
: A security mitigation technique that protects against attacks based on stack buffer overflows.

Complexity Versus Understandability

As a defensive measure, we explicitly chose to implement our CA with limited functionality compared to the full range of options available in the standards (see "Designing Understandable Systems" on page 94). Our primary use case was to issue certificates for standard web services with commonly used attributes and extensions. Our evaluation of commercial and open source CA software options showed that their attempts to accommodate esoteric attributes and extensions that we didn't need led to complexity in the system, making the software difficult to validate and more error-prone. Therefore, we opted to write a CA with limited functionality and better understandability, where we could more easily audit expected inputs and outputs.

We continuously work on simplifying the architecture of the CA to make it more understandable and maintainable. In one case, we realized that our architecture had created too many different microservices, resulting in increased maintenance costs. While we wanted the benefits of a modular service with well-defined boundaries, we found that it was simpler to consolidate some parts of the system. In another case, we realized that our ACL checks for RPC calls were implemented manually in each instance, creating opportunities for developer and reviewer error. We refactored the codebase to centralize ACL checks and eliminate the possibility of new RPCs being added without ACLs.

Securing Third-Party and Open Source Components

Our custom CA relies on third-party code, in the form of open source libraries and commercial modules. We needed to validate, harden, and containerize this code. As a first step, we focused on the several well-known and widely used open source packages the CA uses. Even open source packages that are widely used in security contexts, and that originate from individuals or organizations with strong security backgrounds, are susceptible to vulnerabilities. We conducted an in-depth security review of each, and submitted patches to address issues we found. Where possible, we also subjected all third-party and open source components to the testing regime detailed in the next section.

Our use of two secure zones—one for handling untrusted data and one for handling sensitive operations—also gives us some layered protection against bugs or malicious insertions into code. The previously mentioned CSR parser relies on open source X.509 libraries and runs as a microservice in the untrusted zone in a Borg container.[7] This provides an extra layer of protection against issues in this code.

We also had to secure proprietary third-party closed-source code. Running a publicly trusted CA requires using a hardware security module (HSM)—a dedicated cryptographic processor—provided by a commercial vendor to act as a vault protecting the CA's keys. We wanted to provide an extra layer of validation for the vendor-provided code that interacts with the HSM. As with many vendor-supplied solutions, the kinds of testing we could perform were limited. To protect the system from problems like memory leaks, we took these steps:

- We built parts of the CA that had to interact with the HSM libraries defensively, as we knew that the inputs or outputs might be risky.
- We ran the third-party code in *nsjail*, a lightweight process isolation mechanism.
- We reported issues we found to the vendor.

Testing

To maintain project hygiene, we write unit and integration tests (see Chapter 13) to cover a wide range of scenarios. Team members are expected to write these tests as part of the development process, and peer reviews ensure this practice is adhered to. In addition to testing for expected behavior, we test for negative conditions. Every few minutes, we generate test certificate issuance conditions that meet good criteria, and others that contain egregious errors. For example, we explicitly test that accurate error messages trigger alarms when an unauthorized person makes an issuance. Having a repository of both positive and negative test conditions enables us to perform high-confidence end-to-end testing on all new CA software deployments very quickly.

7 Borg containers are described in Verma, Abhishek et al. 2015. "Large-Scale Cluster Management at Google with Borg." *Proceedings of the 10th European Conference on Computer Systems*: 1–17. doi: 10.1145/2741948.2741964.

By using Google's centralized software development toolchains, we also gain the benefits of integrated automated code testing on both pre-submit and post-build artifacts. As discussed in "Integration of Static Analysis in the Developer Workflow" on page 296, all code changes at Google are inspected by Tricorder, our static analysis platform. We also subject the CA's code to a variety of sanitizers, such as AddressSanitizer (ASAN) and ThreadSanitizer, to identify common errors (see "Dynamic Program Analysis" on page 277). Additionally, we perform targeted fuzzing of the CA code (see "Fuzz Testing" on page 280).

Using Fixits to Harden the CA

As part of our ongoing hardening efforts, we participate in engineering-wide *fixit* exercises. Fixits are a Google engineering tradition where we set aside all of our normal work tasks for a defined period of time and pull together to achieve a common goal. One such fixit focused on fuzzing, where we subjected parts of the CA to intensive testing. In doing so, we found an issue in the Go X.509 parser that could potentially lead to an application crash. We learned a valuable lesson: the relatively new Go library for X.509 parsing may not have received the same degree of scrutiny its older counterparts (such as OpenSSL and BoringSSL) had over the years. This experience also showed how dedicated fixits (in this case, a *Fuzzit*) can shine a spotlight on libraries that need further testing.

Resiliency for the CA Key Material

The most severe risk to a CA is theft or misuse of CA key material. Most of the mandated security controls for a publicly trusted CA address common problems that can lead to such abuse, and include standard advice such as using HSMs and strict access controls.

We keep the CA's root key material offline and secure it with multiple layers of physical protection that require two-party authorization for each access layer. For day-to-day certificate issuance, we use intermediary keys that are available online, which is standard practice across the industry. Since the process of getting a publicly trusted CA included broadly in the ecosystem (that is, in browsers, televisions, and cellphones) can take years, rotating keys as part of a recovery effort after compromise (see "Rotating signing keys" on page 198) is not a straightforward or timely process. Therefore, loss or theft of key material can cause significant disruption. As a defense against this scenario, we mature other root key material in the ecosystem (by distributing the material to browsers and other clients that make use of encrypted connections) so we can swap in alternate material if necessary.

Data Validation

Aside from loss of key material, issuance errors are the most serious mistakes a CA can make. We sought to design our systems to ensure that human discretion cannot influence validation or issuance, which means we can focus our attention on the correctness and robustness of the CA code and infrastructure.

Continuous validation (see "Continuous Validation" on page 174) ensures a system is behaving as anticipated. To implement this concept in Google's publicly trusted CA, we automatically run certificates through linters at multiple stages of the issuance process.[8] The linters check for error patterns—for example, ensuring that certificates have a valid lifetime or that `subject:commonName` has a valid length. Once the certificate is validated, we enter it into Certificate Transparency logs, which allows for ongoing validation by the public. As a final defense against malicious issuance, we also use multiple independent logging systems, which we can reconcile by comparing the two systems entry by entry to ensure consistency. These logs are signed before they reach the log repository for further safety and later validation, if needed.

Conclusion

Certificate authorities are an example of infrastructure that has strong requirements for security and reliability. Using the best practices outlined in this book for the implementation of infrastructure can lead to long-term positive outcomes for security and reliability. These principles should be part of a design early on, but you should also use them to improve systems as they mature.

8 For example, ZLint is a linter written in Go that verifies that the contents of a certificate are consistent with RFC 5280 and the CA/Browser Forum requirements.