Assertion-Carrying Certificates

***Abstract*—Today's TLS certificates are notoriously difficult to augment with new features or even new options under the existing set of features. As a result, the public key infrastructure is unable to quickly evolve to meet new threats, new deployment considerations, and new capabilities. We observe that, fundamentally, certificates are a series of *logical constraints*, limiting what a given principal is able to do. We sketch the design of *assertion-carrying certificates*: certificates that can carry a small amount of code that can dynamically add to these constraints. We present what we believe to be the ideal goals of such a language, and how our initial design in Prolog addresses them. We believe that this modest change to certificates could empower a far more evolvable certificate ecosystem.**

I. Introduction

The importance of the public key infrastructure (PKI) to the success of the Internet cannot be overstated: it is what provides the security in HTTPS, and what ultimately allows users to know with whom they are communicating online. At the core of the PKI are *certificates*: signed attestations that bind a human-understandable name (the "subject name") to a cryptographic public key. Certificates also contain myriad other information, including expiration dates, the *certificate authority* (CA) that signed the certificate, and various extensions that have been added over the years.

Despite its incredible importance, the PKI suffers from a number of issues, chief among them:

It is extremely difficult to *evolve* the PKI. The specification for certificates (X.509) dates back to 1988, and the current version (v3) is from 1999. Updating even simple properties of the PKI's operation is often a multi-year process, owing in large part to the wide install base and variety of players involved [1], [2], [8]–[10].

It is often impossible to *delegate* in the PKI. Any CA trusted to issue certificates can issue certificates for *any* domain. As a result, there are a small number of players with significant power, making them attractive targets and vectors for attack [5]–[7], [11].

We argue that *certificates must be able to evolve* to meet changing threats, unexpected deployment considerations, and new capabilities. This paper puts forth a vision that seeks to achieve this by making certificates more programmable.

II. Assertion-Carrying Certificates

Taking a step back, we observe that certificates fundamentally encapsulate *constraints* on what a principal can do. A certificate constrains the name a principal can claim to be (the subject name), for how long (the expiration date), what its keys can be used for (signing, encrypting, etc.), and so on. Thus, in our view, evolving the certificate ecosystem is a matter of evolving the constraints they can represent.

We propose incorporating small programs into certificates themselves that clients run as a part of their validation process. These programs would be included by the CA who constructs and issues the certificate. As a straightforward example, one could imagine replacing today's expiration ("`NotAfter`") dates with an explicit assertion: `now < NotAfter`.

Prior proposals required significant redeployment and did not provide a specific language [3], [4]. We take a language-centric approach, requiring a modest (we believe) change to certificates. Our primary goals and approaches are:

Non-goal: Turing completeness is not necessary. Our goal is not to be able to run *any* executable from within a certificate. Rather, we envision adding a set of *logical constraints*, which may not even have any side effects.

Our design runs *Prolog* programs inside of an OS sandbox, to limit its system calls, memory consumption, and even run-time (latencies are critical during certificate validation).[1]

Goal 1: Do not broaden the attack surface. This effort is for naught if we ultimately make clients or websites more vulnerable. It is therefore important that our constraint-based language permit formal analysis, be easily testable, and be concise and readable enough to mitigate user error.

Prolog lends itself well to this property because it is incredibly concise. For instance, we re-implemented the certificate validation code from mbed TLS in 82 lines of Prolog.

Goal 2: Never relax constraints. Certificates form a *chain*: the *root* signs an *intermediate* certificate (which can in turn sign more intermediates), which ultimately signs the *leaf* certificate (typically that of a website—which cannot sign any certificates). We envision *each* of these being able to carry their own code; the chain validates if the union of the logical constraints return true. It is imperative that that certificates further down the chain not be allowed to overrule those above them: for instance, constraints set by the root should not be able to be relaxed by any other certificate in the chain.

ACCs achieve this by enforcing that each certificate's code is in its own immutable namespace: certificates can call into others' code, and add new constraints in their own namespaces, but they cannot remove constraints from others'.

Example: Naming constraints. Thus far, we have implemented several new features that have long been debated (but not deployed) by the PKI ecosystem, including *naming constraints* (the assertion that "all certificates farther down the chain must end in the name `.example.com`"), which took 5 lines of Prolog. Name-constrained certificates allow a CA to delegate *some* of its signing power to others.

[1]We have also considered using Datalog (a strict subset of Prolog), but currently opt for Prolog due to its ability to provide namespaces and negation.

References

[1] Abusing bleeding edge web standards for appsec glory, 2016. https://www.dropbox.com/s/63zlhsuhwtwfd12/us-16-Zadegan-Abusing-Bleeding-Edge-Web-Standards-For-AppSec-Glory.pdf?dl=1.

[2] Alexa top 1 million analysis - august 2017, 2017. https://scotthelme.co.uk/alexa-top-1-million-analysis-aug-2017/.

[3] John DeTreville. Making Certificates Programmable. In *First Annual PKI Workshop*, 2002.

[4] Cynthia Dwork and Christina Ilvento. SmartCert: Redesigning Digital Certificates with Smart Contracts, 2020. https://arxiv.org/pdf/2003.13259.pdf.

[5] Dennis Fisher. Final report on diginotar hack shows total compromise of ca servers. https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/.

[6] French gov used fake google certificate to read its workers' traffic. https://www.theregister.co.uk/2013/12/10/french$_{g}ov_{d}odgy_{s}sl_{c}ert_{r}eprimand/$.

[7] Google warns of fake digital certificates issued for its domains and potentially others. https://venturebeat.com/2015/03/23/google-security-temporarily-compromised-by-fake-digital-certificates/.

[8] Intent to deprecate and remove: Public key pinning. https://groups.google.com/a/chromium.org/forum/!topic/blink-dev/he9tr7p3rZ8.

[9] C. Jackson J. Hodges and A. Barth. Http strict transport security (hsts).

[10] Michael Kranch and Joseph Bonneau. Upgrading https in mid-air: An empirical study of strict transport security and key pinning. In *Proc of NDSS*, 2015.

[11] Misissued/suspicious symantec certificates. https://groups.google.com/forum/!msg/mozilla.dev.security.policy/fyJ3EK2YOP8/yvjS5leYCAAJ.

Appendix

In this appendix, we provide an example snippet of code: our implementation of naming constraints. In standard Prolog formatting, this would be four assertions over five lines of code; we have reformatted it for additional readability here.

```prolog
/* Does String end with Suffix */
endsWith(String, Suffix) :-
  string_concat(_, Suffix, String).

/* Is certificate Y a descendant of X */
descendant(X,Y) :-
  signs(X,Y);
  signs(X,Z), descendant(Z,Y).

/* Does the certificate's name end in Suffix */
nameConstrained(Cert, Suffix) :-
  hasName(Cert,Name), endsWith(Name, Suffix).

/* Each descendant X of cert is name-constrained */
forall(descendant(cert,X),
       nameConstrained(X, ".example.com")).
```

Listing 1. ACCs Prolog-based implementation of naming-constraints. Recall that, in Prolog, semicolons denote "or" and commas denote "and".