Chapter 8

PKI: Public Key Infrastructures

> "I see a complex netting of obligations, but within it there is a pyramid of power. No one is truly independent, but as you near the top of the pyramid power increases enormously; however, it is seldom used to its fullest. There are lines of obligation that reach in all directions, upwards, downwards, sideways in a totally alien manner."
>
> Charlie in *The Mote in God's Eye*
> —LARRY NIVEN AND JERRY POURNELLE

8.1 What's a Certificate?

Public key cryptography, as originally described by Diffie and Hellman [1976], seemed simple. Someone uses your public key to encrypt a message to you; you use your private key to decrypt it. However, Diffie and Hellman paid little attention to how communicants acquire each others' public keys, saying only:

> The enciphering key E can be made public by placing it in a public directory along with the user's name and address. Anyone can then encrypt messages and send them to the user, but no one else can decipher messages intended for him.
>
> Public key cryptosystems can thus be regarded as multiple access ciphers. It is crucial that the public file of enciphering keys be protected from unauthorized modification. This task is made easier by the public nature of the file.

> Read protection is unnecessary and, since the file is modified infrequently, elaborate write protection mechanisms can be economically employed.

They do not say where this public directory is, who runs it, how the other party gets access to it, or just what these "elaborate write protection mechanisms" might be. More seriously, they do not conduct a threat analysis. How does everyone agree on or find the proper public directory? Who runs it? Can you trust that party with the "crucial" responsibility of write-protecting the file? How does that party distinguish "unauthorized" from authorized modification? In other words, there are profound systems questions.

Part of the answer was devised by an MIT undergraduate, Loren Kohnfelder, who invented *certificates* [1978]. In the simplest form, a certificate is a digitally signed message containing a user's name and his or her public key. Reality, of course, is more complex; at a minimum, real-world certificates need things like algorithm identifiers.

Today, certificates are generally embedded in a framework known as *public key infrastructure (PKI)*. The Internet Security Glossary [Shirey 2007] defines PKI as "The set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke digital certificates based on asymmetric cryptography." Note carefully that the definition includes "people, policies, and procedures," and not just code. Because of the wide variety of uses of PKI, the semantics of many of these operations are quite complicated; I'll only skim the surface. (For more information on the details of PKI, see [Housley and Polk 2001].)

Complex semantics tends to breed complex syntax; certificates are no exception. Most certificates you will encounter use the X.509 standard [ITU-T 2012], and in particular the Internet's "PKIX" profile [Cooper et al. 2008]. X.509 has complexity in full measure, including highly structured names, highly structured addresses, serial numbers, usage flags, and even corporate logos [Santesson, Housley, Bajaj, et al. 2011; Santesson, Housley, and Freeman 2004], among many other fields. (Actually, originally X.509 wasn't even intended to define general-purpose certificates, but that's another story.)

A more interesting addition to the basic concept are *attribute* fields. An attribute is some characteristic about the certificate holder, attested to by the signer. For people, it might be something like their age. Attributes can be used with or without names. As we shall see, pure attribute certificates are quite useful.

The fundamental questions about certificates are about security: who signs the certificates? Do you trust them? Are they honest? Are they competent, both at procedures (e.g., verifying the holder's identity) and technically (e.g., preventing unauthorized access to their signing key)? Questions like these are at the heart of this chapter.

8.2 PKI: Whom Do You Trust?

When you use a certificate in any way, you are utterly relying on the trustworthiness of its issuer. Understanding the exact mechanisms used today isn't easy, though; the total certificate system architecture is quite complex. Let's look at the pieces of the traditional X.509 setup.

The heart of the certificate system is the *certificate authority (CA)*. A CA does just what its name implies: it issues—signs—certificates. These certificates may be for end users, or they may be for sub-CAs. For end-user certificates, they also indicate (up to a point) the permitted uses for the certificate: encryption, digital signatures, etc. If a sub-CA issues a certificate, the trust question becomes more complex: you have to trust not just the immediate issuer but also every other CA up to the root of the tree. After all, you may trust MyFavoriteInternetCA.com, but the CA certificate it has may have been issued by MI-31.mil.Andromeda, whom you don't trust at all. (They even have a non-existent domain name—the Andromedans are tricky....) Is it the real MyFavoriteInternetCA.com who certified the site you're talking to, or are you being tricked? The root of this tree is called the *trust anchor*, and though it may be turtles all the way down,[1] it has to be trust all the way up.

If there's only one CA in your universe, life is relatively simple. Unfortunately, that isn't the universe most of us live in. Most commercial operating systems, and especially web browsers, come equipped with a very large set of CAs built in. This implies that your vendor trusts them—but do you? Are they (your vendor *and* the CAs) honest? Competent? Most crucially, does their threat model match yours?

The existence of sub-CAs raises another thorny question: what is the permissible scope of activity of the sub-CA? If some company example.com is a sub-CA, it's perfectly reasonable for it to want to issue certificates to its own divisions (e.g., hr.example.com) or employees (e.g., Mary@hr.example.com). Can it legitimately issue a certificate for ARandomBrand.com? What if that corporation is a subsidiary of example.com? What if it was a subsidiary but has since been sold?

More subtly, sometimes an employee speaks for the corporation and sometimes he or she speaks personally. In a paper world, of course, you can't tell whether the person who signed a contract was authorized to do so by her employer. Should certificates embody that sort of authority? Can you tell? You might think that music.example.com was one division of some large media conglomerate, but it turns out that "Music" is the 6,304th most common surname in the United States;[2] perhaps the cert belongs to Mary Music's laptop. ("Lawyer" is #6,309. I think that that's a coincidence.)

1. "Turtles all the way down," https://en.wikipedia.org/wiki/Turtles_all_the_way_down.
2. "dist.all.last," http://www.census.gov/topics/population/genealogy/data/1990_census/1990_census_namefiles.html.

There are special rules for special circumstances. In certificates representing ownership of IP addresses, for example, there are explicit rules for ensuring that certificates contain only address ranges that are subsets of those owned by the issuing CA or sub-CA [Lynn, S. T. Kent, and Seo 2004, Section 2.3]. Of course, the root CA has to have the right to those addresses before it can delegate them. That's a political issue; on the global Internet, those rights are held by the five *Regional Internet Registries (RIRs)*, those rights are assigned to them by the *Internet Assigned Numbers Authority (IANA)*.

Other special rules exist as well. There are standard ways to indicate that a certificate can be used for signing executables, or for email or web encryption. For CA certificates, there is a "Name Constraints" field; this means that a CA trusted to issue certificates for, say, *.example.com can't issue fake certificates for some other company. In general, though, the precise role of a certificate is not obvious, especially to a program.

Sometimes, the policies of a CA matter. CAs are supposed to document their policies in a *Certificate Practice Statement (CPS)*. In the real world, few people even know of the existence of CPSs, let alone try to read them, but since they're often very long and written in legalese it isn't clear that that matters much. More seriously, the party most likely to know about the CPS is the one to whom the certificate was issued, while the relying party—who is most affected by failures—is much less likely to even know of its existence. Let's put it like this: when you decide to do some shopping on the Internet, do you even look to see which CA issued the certificate to the site you're visiting? Do you examine the certificate thoroughly enough to find a pointer to the CPS? Do you then download and study it? Some CAs appear to claim that you're legally required to read their CPS before going to any site that uses it. Here's one from Symantec:[3]

> WHETHER YOU ARE AN INDIVIDUAL OR ORGANIZATION, YOU ("RELYING PARTY") MUST READ THIS RELYING PARTY AGREEMENT FOR USER AUTHENTICATION CERTIFICATES ("AGREEMENT") EACH TIME BEFORE VALIDATING A SYMANTEC-ISSUED USER AUTHENTICATION CERTIFICATE ("SYMANTEC CERTIFICATE") , USING SYMANTEC'S ONLINE CERTIFICATE STATUS PROTOCOL (OCSP) SERVICES, ACCESSING OR USING A SYMANTEC DATABASE OF CERTIFICATE REVOCATIONS OR RELYING ON ANY INFORMATION RELATED TO THE SYMANTEC CERTIFICATE (COLLECTIVELY, "SYMANTEC INFORMATION"). IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT SUBMIT A QUERY AND DO NOT DOWNLOAD, ACCESS, OR RELY ON

3. "Relying Party Agreement for User Authentication Certificates," https://www.symantec.com/content/en/us/about/media/repository/relying-party-agreement-user-authentication.pdf.

How Many CAs Does Your Browser Trust?

Still not convinced there's a problem? Let's look at the list of trusted CAs. As of October 2011, Microsoft listed 320 different root certificates.[a] It had been 321, but one was removed because the CA, DigiNotar, was hacked and its private signing key stolen, allegedly by parties linked to the government of Iran [Galperin, Schoen, and Eckersley 2011] (or maybe the NSA [Schneier 2013]). More than 100 different company names are represented (because of assorted mergers and acquisitions, it's difficult to tell just how many companies are actually involved) from 49 different countries. More than 30 of the certificates are explicitly listed as belonging to agencies of national governments. Does that give you a warm, fuzzy feeling?

Mozilla lists 150 different CAs, from about 60 different organizations. Their database does not have an explicit country indication, but 8 of the CAs are identified as belonging to government agencies.[b]

It's harder to assess Apple's list, since they don't appear to have a single web page; examination of the system certificate file on a computer running Mac OS X "Lion" (10.7.3) on April 12, 2012, shows about 180 different certificates, from at least 30 countries and more than 70 companies.

Note that any of these CAs can create unrestricted sub-CAs, which you can't even see listed in your browser. Any of these organizations can issue certificates for any site on the net.

Is that bad enough? It's worse: some vendors will "helpfully" update your trusted CA list automatically [Microsoft 2009]:

> Root certificates are updated on Windows Vista automatically. When a user visits a secure web site (by using HTTPS SSL), reads a secure email (S/MIME), or downloads an ActiveX control that is signed (code signing) and encounters a new root certificate, the Windows certificate chain verification software checks the appropriate Microsoft Update location for the root certificate. If it finds it, it downloads it to the system. To the user, the experience is seamless. The user does not see any security dialog boxes or warnings. The download happens automatically, behind the scenes.

What you're supposed to do instead is to add unwanted certificates to the "untrusted" list, through a fairly ghastly user interface.

---

a. "Windows Root Certificate Program — Members List (All CAs) — TechNet Articles — United States (English) — TechNet Wiki," http://social.technet.microsoft.com/wiki/contents/articles/2592.aspx.

b. "Included Certificate List," http://www.mozilla.org/projects/security/certs/included/.

> ANY SYMANTEC INFORMATION. IN CONSIDERATION OF YOUR AGREEMENT TO THESE TERMS, YOU ARE ENTITLED TO USE SYMANTEC INFORMATION AS SET FORTH HEREIN. AS USED IN THIS AGREEMENT, "SYMANTEC" MEANS SYMANTEC CORPORATION OR ANY OF ITS SUBSIDIARIES.

The shouting caps are all theirs. Note what it says: before "validating" or "relying" on information in the certificates they issue, you "must read this" agreement. Your browser did tell you that, right? No? I guess you're not allowed to buy things online.

All of this—the certificates, the trust anchors, the delegation rules, the revocation mechanisms (see Section 8.4)—compose what we know as PKI. PKI is the subject of a great deal of angst, fear, misinformation, disinformation, and downright mythology. Unfortunately, the foci of all of this Sturm und Drang are generally the complexity and security issues. While these are indeed of concern, we now see the crucial limitations of certificates for most people:

- It is rarely clear to system administrators or developers which CAs are trusted for given applications. It is almost never clear to end users.
- It is rarely clear to anyone what a given certificate's intended use is.
- It is almost never clear how trustworthy or competent a CA is. Many years ago, Matt Blaze observed that a commercial CA would protect you from anyone from whom they wouldn't take money [2010]. This is a crucial point: if you trust a system's built-in CAs, you are in effect trusting some unknown set of third parties to vouch for someone else's identity (or attributes) and making access control decisions based on these third parties' opinions.

It helps here to reason by analogy. Suppose someone who claims to be an employee wants to walk into your building. Rather than showing an employee ID card, they instead pull out what appears to be a credit card in a real employee's name; this credit card was issued by one of several hundred banks you've never heard of, possibly up to and including the Bank of San Serriffe.[4,5] Do you think they should be admitted to your building? Would you accept that as a login credential for your computer systems?

Any of these CAs can issue a certificate for any web site in the world—and your browser will accept it.

4. "San Serriffe," http://www.museumofhoaxes.com/hoax/archive/permalink/san_serriffe.
5. "Knuth: The Bank of San Serriffe," http://www-cs-faculty.stanford.edu/~knuth/boss.html.

8.3 PKI versus PKI

Although, as I've shown, the standard Internet-wide PKI is unacceptably insecure, most of the tools and pieces can be used quite securely if we can reverse the three big problems listed above. That is, if we can construct a scenario in which everyone knows exactly who can issue a certificate and what the purpose of that certificate is, and if the issuer can be trusted to an extent commensurate with the resource being protected, we will have a secure (or secure enough) system, *while using the same software, syntax, and so on*. In particular, if we give up on the notion of the One True PKI and the One True List of CAs, and instead have a CA per access control point, we can reap most of the benefits of public key cryptography while avoiding the pitfalls. I call this concept PKI, to suggest small-scope CAs, rather than the large-scope CAs of a traditional PKI. To expand on the credit card analogy, enterprises generally issue their own employee credentials that are used to enter the building, rather than deferring to the bank. Furthermore—and I'll expand on this point—employee ID cards function as *authorization* tokens even more than they do as identification devices. It's your possession of the card (possibly as authenticated by your picture—a biometric—and perhaps knowledge of a PIN) that gets you in the door, rather than your name and address.

The key insight is that every function that requires (or would benefit from) use of certificates should have its own PKI. Thus, an IPsec gateway would issue its own certificates; these would be distinct from the certificates issued by, say, the corporate email service for use in encrypting and authenticating messages. Similarly, the outside company that handles procurement would issue a sub-CA certificate to the enterprise; this CA would in turn certify employees who are authorized to buy things.

Note how this solves the problems described in the previous section. The IPsec gateway CA is trusted for IPsec; your web browser won't believe certificates that it issues for, say, Amazon.com. The purchasing certificate from ReallyNiceCorporateToys.com comes from a CA that has the name ReallyNiceCorporateToys.com; no one will think it should let you use IPsec.

The trustworthiness issue is the most important distinguisher. By definition, PKI certificates are issued by the party that is entitled to grant access to some resource. It does not have to be—and is not—trusted to grant or withhold access to anything else. Furthermore, given that such parties are handing out access credentials, from a security perspective it does not matter whether these credentials are key pairs, passwords, or special thought symbols that are sensed by a magic crystal under the users' keyboards. The actual technologies used may have differing security properties, but the fact that public key technologies are used does not change the powers of the grantor. It is certainly possible to overload the meaning of a PKI certificate, just as (in the United States) drivers' licenses

are used to gain access to airplanes and alcohol; the trick (and it's at least as much organizational as technical) is to resist the temptation and do things properly. Use these certificates only for the purpose for which they were issued.

A consequence of this is that running a PKI does not imply a need for excessive complexity or security. In describing a simplified approach to IPsec configuration [Srivatsan, M. Johnson, and Bellovin 2010], we wrote:

> Public key infrastructures (PKIs) are surrounded by a great mystique. Organizations are regularly told that they are complex, require ultra-high security, and perhaps are best outsourced to competent parties. Setting up a certificate authority (CA) requires a "ceremony", a term with a technical meaning [Ellison 2007] but nevertheless redolent of high priests in robes, acolytes with censers, and more. This may or may not be true in general; for most IPsec uses, however, little of this is accurate. (High priests and censers are definitely not needed; we are uncertain about the need for acolytes...)
>
> Much of the mystique is due to the general-purpose nature of PKIs and certificates. If a certificate is intended to attest to a person's identity, a lot of process may be necessary. The real danger from a compromised root key comes from the attacker's ability to create arbitrarily many fraudulent credentials.

This is crucially important: a PKI installation need not be surrounded by any more security than any other credential-issuing system. Some years ago, I heard a presentation from a PKI vendor; the speaker stressed how "court-certified videographers" recorded all root key ceremonies. Would you do the same when setting up your payroll system? Your ID card system? Of course not—but why is it needed for certificate issuance? In multi-organization situations, it could be defended as a mechanism to ensure trust; internally, that isn't needed.

A PKI avoids the central philosophical contradiction of general-purpose PKIs: the CA is not authoritative for the name space concerned. This means that there are two different entities that have to vouch for someone's identity, the actual name space owner—on the web, that's whoever controls the DNS name—and the CA. Much of the bureaucratic overhead of large-scale PKIs is dedicated to ensuring that their decisions match those of the name space owner.

Running a PKI instead of using a KDC or even, in some cases, a password file has another, more subtle advantage: successful attacks against your centralized infrastructure are less damaging. Secret keys and password files must be protected against disclosure. In other words, you have a confidentiality problem. By contrast, CAs need only authentication, a simpler problem. (Recall Table 7.1 and Figure 7.4: public key-based authentication is the most secure type available.)

Apart from hype and myth-making, there's another reason that many organizations refrain from issuing their own certificates: it seems like a very hard thing to do. There's some truth to that, albeit for a very bad reason. Let's face it: many of the readily available certificate-minting software packages are not only not user friendly, they're downright user hostile. Much of the complexity comes from asking for information that is generally irrelevant and often incomprehensible. Too many fields don't need to be user settable in 99.99% of cases. Do you really need to know (or want to know) what the difference is between the "keyUsage" and "extendedKeyUsage" fields? A vendor of cryptographic software is almost certainly more qualified than a typical organization to pick, say, appropriate algorithms, key lengths, expiration periods, and so on, but the poor soul who has to use the software has to read through (and understand) all of these options before concluding that the default values can be left alone. Other fields, such as Organizational Unit or City, might be necessary for identity certificates; they're generally irrelevant for the certificates I'm advocating, which are *authorization certificates*. That is, they give access to some resource, but by intent they give access to the holder of the corresponding private key. By contrast, identity certificates are used for authentication; after that, the user's name is matched against some form of access control list.

The ability to do something like this naturally suggests the idea of a company issuing certificates for its own internal web sites, rather than buying them from a commercial CA. It's not a bad idea, but alas it is probably not worth it. Yes, it's good that your own internal IT department is the one attesting to the identity of, say, the Payroll web site. Alas (and as described above), the risks arise because all of the other CAs your browsers trust can also issue such certificates; having your own CA doesn't change that. You could try deleting them, but then you're faced with the challenge of keeping up with the deletions, that is, knowing every machine in the company (including all of the mobile devices people are using), knowing what browsers are in use, knowing how to delete CAs from all of these, actually having the authority to do so, keeping up with "helpful" vendor functionality that restores CAs (see the box on page 153), and so on. An organization with a highly centralized, all-controlling IT group may be able to accomplish this (except, perhaps, working around that excessive helpfulness), but excessive rigidity has its own disadvantages [Perrow 1999].

Some companies do, in fact, do this, for a somewhat different reason: their firewalls want to inspect all traffic, even if it's HTTPS-protected. To do that, they use local CAs that issue fake certificates for any web site; this permits the firewall to decrypt and then reencrypt all traffic. There are interesting liability questions if this is done to, say, banking web sites, but of course many organizations bar all non-work usage. (On the other hand, I once worked for an organization that explicitly based its computer usage policies on

Deuteronomy 25:4: "You shall not muzzle an ox while it is threshing." A modest amount of personal usage was explicitly permitted.)

There's one more bad idea in this space that should be disposed of. Some people will get the notion that they should delete all CAs that have issued certificates for Facebook and other sites on which employees "waste" time. It won't work. People who want to get to such sites will just click through all of the bloodthirsty warnings their browsers will display; you'll only succeed in training people to ignore security pop-up messages.

If you don't want to rely on a PKI or a PKI, there's another way to get a great deal of safety: *key continuity* (also known as *certificate pinning* or *key pinning*). Key continuity relies on a simple concept: public keys rarely change. Accordingly, applications can record the key sent by a peer; if there's a difference on successive connections, it's quite possible that some evil party is trying to play nasty games. On the other hand (and as shown in the error message in Figure 8.1), key changes can also occur for perfectly benign reasons. In fact, this situation *will* occur. Knowing how to deal with it—and training users how to react *when* they see such errors—is crucial; you do *not* want your users to become acclimated to clicking through error messages. A variant checks whether the issuing CA has changed; while less likely to generate false alerts—organizations don't change their CA vendors that often—such changes are by no means impossible. Indeed,

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now
(man-in-the-middle attack)!
It is also possible that the RSA
host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
c5:10:e6:70:18:65:22:6f:48:71:26:26:3f:6d:2b:07.
Please contact your system administrator.
Add correct host key in /Users/smb/.ssh/known_hosts to get
rid of this message.
Offending key in /Users/smb/.ssh/known_hosts:150
RSA host key for some.host has changed and you have requested
strict checking.
Host key verification failed.
```

Figure 8.1: A key continuity failure message.

making it hard to switch CAs would create a serious vendor lock-in problem. The key or CA change problem, and the question of protecting initial contacts, has limited the use of key continuity; to my knowledge, the only popular non-web application that uses it is ssh. Still, it is a powerful tool in the right hands.

The IETF has recently created an HTTP extension for key pinning [Evans, Palmer, and Sleevi 2015]. The new header fields defined can specify the length of time for which a pin should remain active. In addition to permitting pinning of the certificate itself, it permits pinning to a particular CA, thus allowing for easy issuance of new certificates for the site. In addition, there is provision for a reporting URL, so that the web site owner can be informed of spoofed certificates. This will work well for some sites; it remains to be seen how well it will work in general.

Another way to look at the PKI problem is that it has created a technical decoupling of trust from user-accessible concepts. That is, users try to connect to a service (e.g., the web) on some specific domain. They thus implicitly trust the DNS to give them the proper IP address. If we add DNSSEC [Arends et al. 2005a; Arends et al. 2005b; Arends et al. 2005c] to the mix, this is a reasonably secure process. With today's PKI, though, trust is coming from the collection of root CAs. If the sites' certificates (or the hashes thereof) were stored in the DNS and protected by DNSSEC, there would be no need to trust the CAs; each site could control its own cryptographic fate. This is the approach being taken by the IETF's *DNS-based Authentication of Named Entities (DANE)* working group [Barnes 2011; Hoffman and Schlyter 2012]; however, it is crucially dependent on DNSSEC, which has not yet seen widespread adoption. It is too soon to predict what will happen here.

Earlier, when I spoke of PKI CAs issuing certificates to users, I know that some of you were practically jumping out of your seats and raising your hands to ask about usability. It's a serious issue: in a world of multiple devices per person, how can normal users securely store and manage a large collection of private keys? Fortunately, with a bit of software assistance, it's not a show-stopper.

Recall that in Section 7.3, I discussed password managers. The same concept (and much of the same code) can be used to handle private keys, while encrypting them for storage in some convenient place, for example, a cloud provider. Again, if a private key is simply an access token for a single service, it does not need to be protected more strongly than a password would be for the same service. It would be nice if there were secure, convenient, portable key storage devices that worked well with our laptops, phones, tablets, smart light switches, and the like, especially for high-value keys, but such tokens have been just around the corner for well over a decade.

DANE versus Certificate Transparency

Some people have practical or philosophical objections to DANE. DNSSEC responses are large, there are complex technical issues hindering its deployment, and there is serious reason to wonder whether DNS registrars understand the security challenges involved in running a CA-like service. One alternative proposal is a Google proposal, *Certificate Transparency (CT)* [Laurie, Langley, and Kasper 2013]: every CA would log all of the certificates it issue, thus permitting browsers to notice if two different CAs had issued a certificate for some site. For that matter, any company that wishes could monitor the various CT logs to see whether certificates appear in inappropriate places.

The trouble with CT is that it requires universal compliance; otherwise, there's no way to protect against a non-CT CA issuing the bogus certs. To date, most CAs have indicated that they are not enamored of CT, quite notably including Symantec, one of the major CAs.[a] It is unclear whether there will be sufficient participation for it to fly, though it has already detected one incident [Goodin 2015d].

The security concerns about registrars are quite plausible. With today's CA structure, as problematic as it is, someone attacking a secure web site has to subvert two independent mechanisms: the routing or DNS entry that controls where the traffic goes, and the CA. With DANE, anyone who can seize control of a company's DNS entry—and that has happened, even to security-savvy firms [Edwards 2000]—can replace the certificate. Is this better or worse than CT? A lot depends on your threat model, but there isn't room in the Internet for two solutions to the PKI conundrum.

a. "Upcoming changes to Google Chrome's certificate handling," https://cabforum.org/pipermail/public/2013-November/002336.html.

8.4 Certificate Expiration and Revocation

Regardless of whether we use a PKI or a PKI, certificates do not last forever. After some time interval specified in the certificate (i.e., set by the CA), certificates expire. Alternatively, they can be revoked for any number of reasons, including if the private key is believed to have been compromised. Expiration and revocation seem straightforward. They're not, but before diving into the complexity, let's look at how certificates die.

Certificates can become invalid ("die") in one of two ways. First, they can expire; all certificates include an expiration date, after which they may not be used. Second, they can be revoked, that is to say, explicitly declared to be invalid. The latter may be done

because of private key compromise, suspected or actual misbehavior by the holder of the private key, or fears for the strength of the cryptographic algorithms used.

There are three reasons for expiration. First, there is the sense (often a vague sense) that after a certain period of time, the likelihood of an undetected compromise of the private key has become unacceptably high. The mathematics of this policy are impeccable. Suppose that the probability of a compromise during a given time interval is p; further suppose that the intervals are independent. Obviously, then, the probability of security—that is, of no compromise—after n intervals is $(1-p)^n$. Pick your trustworthiness probability threshold t and solve, getting $n = \frac{\log t}{\log 1-p}$; it's nice, simple, and mathematical. It's also a useless exercise, since no one has any good idea what p might be. A few years is a common choice, but the mathematical basis for that is nil.

The second reason for certificate expiration is that algorithms age. Back in the dawn of time when the web was young, CAs commonly issued certificates with 1,024-bit keys and MD5 as the hash algorithm. Both are now believed to be insecure; suitable choice of an expiration date protects against that. This, too, seems like an impossible decision—how can you know when an algorithm will be cracked?—but in reality, modern algorithms do not fail all at once. Generally speaking, cracks will show up years in advance. To give just one example, signs of weakness in MD5 were noted as early as 1996 [Dobbertin 1996], well before the 2004 crack [X. Wang et al. 2004]. People realized this, even without hindsight; to give just one example, Bill Cheswick, Avi Rubin, and I warned about it in *Firewalls* [2003, p. 347]. A certificate lifetime of a few years should allow enough time to change; your credentials based on insecure algorithms will probably expire and be replaced before the problem becomes critical.

Finally, certificates expire to ease bookkeeping with respect to revocations: there's no apparent need to keep track of the revocation status of an expired certificate because it is a priori invalid. As we shall see, though, that benefit is more illusory than real.

A party accepting a certificate can check for revocation in two different ways. The older mechanism uses a *Certificate Revocation List (CRL)* [Cooper et al. 2008], a file of revoked certificates signed by the issuing CA; the URL of this list is included in certificates. Those of a certain age may recall store clerks looking up credit cards in a book-format blacklist; conceptually, this is the same thing, save that CRLs include the time of the next update to provide warning of stale revocation lists. (Just what a relying party should do if a new CRL doesn't arrive on time is a difficult question. Clearly, something is wrong; quite possibly, a denial of service attack has been launched to prevent folks from learning of newly revoked certificates. Equally clearly, rejecting all presented certificates just because you can't retrieve a new CRL is very unlikely to be correct.)

The other way to check the validity of a certificate is via a network connection using *Online Certificate Status Protocol (OCSP)* [Myers et al. 1999]. The obvious analogy,

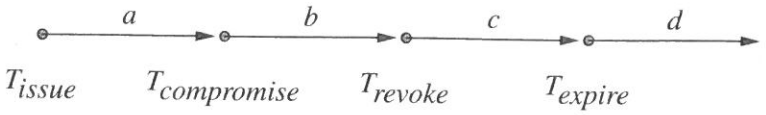

Figure 8.2: Timeline of certificate compromise, revocation, and expiration.

of course, is a modern credit card terminal. Don't stretch that analogy too far; credit card numbers are not self-checking the way certificates are. OCSP is used to verify the continuing validity of a certificate, rather than whether it was ever valid. OCSP can return Valid, Invalid, or Unknown status codes; again, the question of what to do if the OCSP server is unavailable or returns Unknown is a difficult one.

Intuitively, OCSP seems "more secure," in that it reduces the time between key compromise and effective revocation. The actual benefit, though, is much less. Consider the timeline in Figure 8.2. The effect of OCSP is to shrink interval b, the time between compromise and a "don't trust this" signal. Almost always, though, the time between $T_{compromise}$ and *discovery* of the problem is much longer than the time between discovery and effective revocation. Yes, OCSP reduces the effort needed for effective revocation, but realistically that's almost always a small percentage of the interval b.

Furthermore, the very concept of certificate death isn't as simple as it appears. Let's look at expiration first. It seems obvious—after a certain date D, the certificate may not be used—but "used" for what? Suppose that someone encrypts and signs a file at date $D-1$, when the certificate is still good. Are you allowed to verify the signature at $D+1$? Obviously, you should be; anything else would be absurd. What should happen, though, if the sender created the file at $D+1$? That shouldn't happen—but what should you, the recipient, do if it does, and how can you tell if you don't have access to the sender's clock? (The problem of clock synchronization between the sender and the recipient is an entirely separate can of worms.)

Digital signatures present an even more complex challenge. The point of a digital signature is to provide proof that some individual (more accurately, some private key) created a particular file or message. However, one reason why certificates expire is to deal with unsuspected compromise of its private key. How should a recipient handle a message that was signed with a now-expired key? The timestamp in the message itself may indicate that it was signed before the expiration date—but can you trust the signer's timestamp? After all, that value might have been set by the enemy. Equally problematic, how do you prove to a third party, such as a judge, just when the message was signed? There are schemes for timestamping documents—see, for example, [Haber and Stornetta 1991a; Haber and Stornetta 1991b]—but they're rarely used, at least in the United States.

Ultimately, the issue becomes a very deep question: when does your *reliance* on a signature expire? That is, when should you no longer take actions based on a message

that was signed with some particular key? A case in point is a software package, such as a device driver, that was signed by the vendor. Depending on the package, a considerable amount of time may elapse between signature time and the time the signature is checked. A lot can happen between those two events.

Let's first consider an easier situation: receiving a digitally signed email message, perhaps containing an assent to a contract. There are four parties to consider: the CA, the signer, the relying party, and a judge who may be asked to rule on disputes. Look again at the timeline in Figure 8.2; the certificate was created at T_{issue}. If the message is received during interval a, clearly all is well; no compromise has yet happened. Interval b is the danger period: the private key has been compromised, but the certificate has not yet been revoked. The size of the interval depends on two different factors: how much time has elapsed between the actual compromise and its detection, and how long the various processes take to actually revoke the certificate.

Interval c is safe *if* the relying party actually checks for revocation. As we have seen, different revocation schemes have different properties here. Depending on the scheme and various environmental considerations, T_{revoke} can move to the right, effectively lengthening interval b. (Revocations do carry an explicit timestamp, but the meaning will vary; it may be the time of compromise as determined by the certificate holder, or it may be when the revocation was requested. You have to check the CPS for details.)

Interval d is unambiguous (everyone checks for expiration) in the sense that you won't trust a compromised certificate, though the exact value of T_{expire} depends on the clock skew between the CA and the relying party.

All this seems simple, but it's not. $T_{compromise}$ is uncertain and often unknowable; was the message signed during interval a or interval b? For that matter, suppose the signer regrets agreeing to the contract and deliberately leaks the key, but claims that it was available earlier? Is the signature genuine or not? What then? This matters if there's a dispute: who actually signed the message, the nominal signer or an attacker who has the compromised private key? Ultimately, a judge may have to decide.

The problem intended to be solved by digital signatures was described this way by Diffie and Hellman in their original paper [1976]:

> In current business, the validity of contracts is guaranteed by signatures. A signed contract serves as legal evidence of an agreement which the holder can present in court if necessary.

and

> That is, a message may be sent but later repudiated by either the transmitter or the receiver. Or, it may be alleged by either party that a message was sent when in fact none was. Unforgeable digital signatures and receipts are

needed. For example, a dishonest stockbroker might try to cover up unauthorized buying and selling for personal gain by forging orders from clients, or a client might disclaim an order actually authorized by him but which he later sees will cause a loss. We will introduce concepts which allow the receiver to verify the authenticity of a message, but prevent him from generating apparently authentic messages, thereby protecting against both the threat of compromise of the receiver's authentication data and the threat of dispute.

Unfortunately, the issue of key compromise has dimmed the luster of Diffie and Hellman's solution. The crucial advantage provided by digital signatures, non-repudiation, vanishes in the face of a deliberate key leak by the signer. This means that ultimately, the authenticity of a signature is a factual question, of the sort commonly handled by courts. Unfortunately, in this situation the crucial evidence—log files, forensic examination of the purported signing computers, and so on—is of a sort rarely handled well by courts and juries; the material is too technical. Your best defense is logging (Section 16.3) and insistence on logging by the other party; ideally, such logs are kept by an independent party. Think notary publics, moved to the digital world.

Life is even more complicated in other digital signature scenarios, where the actual dependency is considerably later than receipt of the signed message, or where "receipt" is not obvious. Consider, for example, a digitally signed device driver, written onto a CD and packaged with the hardware. The real dependency of the signature is not when you buy the device, but when the CD was burned (though of course you don't know whether the signature was checked at that point). A compromise later is quite irrelevant; no hacker, no matter how good, can affect an already created CD. (On the other hand, the Andromedans can launch a *supply chain attack* and tamper with nominally sealed boxes of hardware, substituting their own CD for the genuine one. They've done it in the past.) Sure, you can check whether the certificate has been revoked before you install the device driver, but that isn't a meaningful operation unless you're somehow informed that $T_{compromise}$ was before the CD was created—and of course you can't know that. Note carefully that none of this analysis applies to downloaded device drivers; for those, the situation is similar to the email situation. After all, a clever attacker can easily replace a signed device driver on a vendor's web site.

Generally, attackers aren't trying to tamper with software you know you're installing. Rather, they're trying to take advantage of the implicit *authorization* that some operating systems attach to the *authentication* of files signed by certain CAs. That is, they'll silently install code if the file is signed by a certificate issued by any of a vast number of parties. Stuxnet took advantage of this [Falliere, Murchu, and Chien 2011; Zetter 2014]; there have been other reports of signed malware using the same technique [Bijl 2011; Goodin 2012b; Hypponen 2011; IIJ 2012] as well as at least one accidental key compromise

Are Digital Signatures Legally Binding?

The formal status of a digital signature—that is, of the output of a series of calculations that can only be performed or verified by a computer—is of course a legal matter, not a technological one. In the United States, such signatures are binding, both as a matter of common law and as explicitly codified in the *Electronic Signatures in Global and National Commerce Act* of 2000.[a]

A signature "whether electronic or on paper, is first and foremost a symbol that signifies intent" [Smedinghoff and Bro 1999]. Identifying the signer and ensuring the authenticity of the signed document are described as "secondary purposes." A legal opinion from the Comptroller-General [1991] notes that "Because of its uniqueness, the handwritten signature is probably the most universally accepted evidence of an agreement to be bound by the terms of a contract... Courts, however, have demonstrated a willingness to accept other notations, not necessarily written by hand." In general, then, there is no reason that a digital signature cannot be used.

However—this is, as noted, a legal matter, and there are often exceptions, caveats, and different rules in different jurisdictions. Before assuming that a digital signature is legally binding, consult your own attorney or logomancer.

a. "Public Law 106-229," http://www.gpo.gov/fdsys/pkg/PLAW-106publ229/content-detail.html.

[Goodin 2015c]. As noted in the box on page 82, there is speculation that 512-bit RSA keys were factored in at least one case; with Flame, a new cryptanalytic technique—applied to certificates using MD5, which the CA should not have accepted—was used. In the other cases, it seems more likely that the private key was stolen.

Looking at things more formally, what we really need to consider is the relationship between T_{revoke} and T_{rely}, the time when you *use* the certificate. Ideally, of course, $T_{rely} < T_{revoke}$. The complexities discussed above occur when $T_{revoke} < T_{rely}$; problematic uses include file installation, showing the signature to a judge, and more. Absent more information, dilemmas are inescapable. That, however, points to a solution: more information.

If the digital signature includes a trustworthy timestamp (this will generally be the case for signed software; if you don't trust your vendors to tell the truth about the time, why are you installing their code?), this value can be compared with $T_{compromise}$ when it is learned. (Of course, if it's a fraudulent software package, it *isn't* from the vendor you trust.) If vendors would do things like publish a time-stamped list of signatures they've

created, you could compare your signed files with that list, and with $T_{compromise}$ if and when you learn of it. (Some companies prefer not to reveal that they've been hacked [Yadron 2014].) I know of no vendors who create such lists, and few who publish the results of forensic examinations, but perhaps that will change. Better yet, the vendors could let an outside party maintain the list; that's better for things that someone may want to show to a judge. (If done properly, publishing a signature list doesn't reveal any sensitive information. The list doesn't have to have the signature itself, let alone the file being signed; rather, it can be the cryptographic hash of the actual signature.)

Now consider what happens if $T_{expire} < T_{rely}$. Is using the certificate at that time safe? As before, there isn't enough information to decide; what we really need to know is the relationship of $T_{compromise}$ to $T_{signature}$. Conceptually, this means that one might want to revoke an expired certificate, if the compromise isn't detected until much later. This can happen; see [Naraine 2012] for one example. The existence of a revocation message is an explicit statement of danger, rather than the more generalized feeling of concern that expiration times are intended to handle. Think of it this way: certain diseases are more likely to occur as we age, even though they can occur earlier. However, this general warning—when you reach age X, get tested for such-and-such—is very different than a doctor looking at some test results and delivering the bad news. When there's bad news—when a certificate is known to have been compromised—that fact has to be communicated explicitly, so that you can take appropriate action.

What it comes to is that revocation is rarely used or usable on the Internet. The semantics are unclear, and the very different models for what it means to "use" a certificate means that the effects of such an action are often unpredictable.

One more point bears mentioning: none of the conceptual problems with revocation are related to the difference between PKI and PKI. Those have to do with trust patterns; revocation suffers from complex semantics. Moreover, the complexity is inherent in the problem statement; they are not an artifact of current designs. A PKI limits the damage from a compromised key; it doesn't change the very difficult reasoning about what to trust and when.

8.5 Analysis

What possible changes might affect the recommendations of this section?

The heart of the web PKI problem is the "let a hundred CAs bloom" approach of browser and OS vendors. It seems unlikely that they will change their policies; unless and until either certificate transparency or DANE are deployed, there are no secure alternatives for initial contact. Besides, this is The Way Things Are Done; we are dealing with the Deity of Inertia. Finally, the political implications of having only one root CA—one all-

powerful entity that decides who can and who cannot do secure web interactions—make it an extremely unacceptable alternative.

Key continuity as an add-on is relatively simple to add; however, the key change or CA change scenarios must be dealt with. This is a challenging problem in user experience design. A good solution would present users with comprehensible information about the old and new CAs. I suspect that a perfect solution is an intractable problem; even understanding the question "do you want to trust this change in certificate authorities?" requires a far deeper understanding of PKI than most users have, want to have, or should have. There are browser add-ons (e.g., Certificate Patrol for Firefox) that provide such functionality for sophisticated users; indeed, key continuity checking is how the DigiNotar hack was detected. On the other hand, the number of (apparent?) false alarms is so great that I've disabled it on my computers.

Google has implemented key continuity in Chromium, its open-source operating system and browser. However, the feature announcement came with a warning [Evans 2011]:

> You can now force HTTPS for any domain you want, and even "pin" that domain so that only a more trusted subset of CAs are permitted to identify that domain.
>
> *It's an exciting feature but we'd like to warn that it's easy to break things! We recommend that only experts experiment with net internals settings.* [emphasis in the original.]

Their own sites are protected by default, and we can assume that they would push out an update in advance of any change in their own certificates or CAs. That isn't a solution that is generally applicable.

It would be nice if users could easily download a different collection of CAs from some source that they themselves trust. Enterprises would like this; it would permit easy tailoring of the list to include corporate CAs. There is a downside, though; repressive governments could use it to insert their own CA in the list, at least for browsers distributed within their countries. It also poses a new problem: how is that download to be authenticated? Most likely, every browser vendor would use its own, hard-wired CA to protect such downloads; they would then issue certificates to anyone who wanted to supply a CA list, being careful only to verify the identity of the supplier. There is no strong need to verify their trustworthiness, at least if you assume that people who download the MI-31 CA list are only doing so voluntarily.

There are two other models of certificates that bear mentioning, the *web of trust* and *simple public key infrastructure (SPKI)*. Both are significant steps away from the hierarchical, name-based approach of traditional certificate authorities.

The web of trust, used by the *Pretty Good Privacy (PGP)* mail encryptor [S. L. Garfinkel 1995; Lucas 2006; Zimmermann 1995], certifies name/key bindings via an ar-

bitrary directed graph rather than a tree. There are no keys specifically designated for use by CAs; rather, every key is simultaneously usable for certificate signing and for actual data encryption or signing.

The advantage of the web of trust is that no infrastructure is necessary to start using it; you and your friends can generate keys and sign each others', and you're off and running. You don't have to worry about Andromedan-run CAs; you're trusting your friends. On a small scale, it works well; you can even view it as a form of PKI. Problems appear when you need to make more than one hop. You probably trust your friend to vouch for her friends' identities, but do you trust them as much? Their friends, whom you don't know even at one remove? Have you ever met a friend of a friend who seemed rather sketchy to you, enough so that you wondered why your friend associated with such a person? Scale that up a little and it becomes obvious that trust drops off very rapidly with distance.

Revocation is even more problematic in web-of-trust systems than in hierarchical CAs, since there is no single CRL to consult. There are some well-known PGP key servers, but these are generally used to obtain keys, not to revalidate currently stored keys.

SPKI [Ellison 1999; Ellison et al. 1999] takes a different, even less conventional approach to certificates: it's based solely on authorization, rather than identity. The certificate may contain the name of the putative holder, but that's more a convenience; presentation of the certificate gives the authorization information for the holder, and the private key is (of course) used for authentication. The name is never looked up in any sort of access control list. This decision was made for a number of reasons, most notably because of the difficulty of constructing a single global, unique name space. SPKI also includes the notion of delegation, and a number of set-theoretic operations on collections of certificates to decide whether a particular one provides authorization for a given service; see [Ellison et al. 1999] for details. It's an interesting model, and it has been used in a few situations; it's unclear, though, how well it would function at Internet scale.

It is important to realize that the thorny problems in this chapter—the need to trust many CAs, and the meaning of revocation—are conceptual problems that are inherent in the overall solution space. Simple changes in technology, such as a different hash function or switching to elliptic curve signatures, don't affect them at all; these problems are not susceptible to easy technical fixes. It will take major breakthroughs to find fundamentally different solutions, and a multi-year effort to deploy them.