CHAPTER 20

PKI Practicalities

In practice, if you need a PKI, you will have to decide whether to buy it or build it. We'll now discuss some of the practical considerations that occur when designing a PKI system.

20.1 Certificate Format

A certificate is just a data type with multiple required and optional fields. It is important that the encoding of a particular data structure be unique, because in cryptography we often hash a data structure to sign it or compare it. A format like XML, which allows several representations of the same data structure, requires extra care to ensure that signatures and hashes always work as they should. Although we dislike their complexity, X.509 certificates are another alternative.

20.1.1 Permission Language

For all but the simplest of PKI systems, you really want to be able to restrict the certificates that a sub-CA can issue. To do that, you need to encode a restriction into the sub-CA's certificate, which in turn requires a language in which to express the key's permissions. This is probably the hardest point of the PKI design. The restrictions you are going to need depend on your application. If you can't find sensible restrictions, you should rethink your decision to use a PKI. Without restrictions in the certificates, every sub-CA effectively has a

master key—and that is a bad security design. You could restrict yourself to a single CA, but then you would lose many of the advantages of a PKI over a key server system.

20.1.2 The Root Key

To do anything, the CA must have a public/private key pair. Generating this pair is straightforward. The public key needs to be distributed to every participant, together with some extra data, such as the validity period of this key. To simplify the system, this is normally done using a *self-certifying certificate*, which is a rather odd construction. The CA signs a certificate on its own public key. Although it is called self-certifying, it is nothing of the sort. The name self-certification is a historical misnomer that we are stuck with. The certificate doesn't certify the key at all, and it proves nothing about the security properties of the key, because anyone can create a public key and self-certify it. What the self-certification does is tie additional data to the public key. The permission list, validity period, human contact data, etc., are all included in the self-certificate. The self-certificate uses the same data format as all other certificates in the system, and all participants can reuse the existing code to check this additional data. The self-certificate is called the *root certificate* of the PKI.

The next step is to distribute the root certificate to all the system's participants in a secure manner. Everybody must know the root certificate, and everybody must have the right root certificate.

The first time a computer joins the PKI, it will have to be given the root certificate in a secure manner. This can be as simple as pointing the computer at a local file or a file on a trusted Web server, and telling the machine that this is the root certificate for the PKI in question. Cryptography cannot help with this initial distribution of the root certificate, because there are no keys that can be used to provide the authentication. The same situation occurs if the private key of the CA is ever compromised. Once the root key is no longer secure, an entirely new PKI structure will have to be initialized, and this involves giving every participant the root certificate in a secure manner. This should provide a good motivation for keeping the root key secure.

The root key expires after a while, and the central CA will have to issue a new key. Distributing the new root certificate is easier. The new root certificate can be signed with the old root key. Participants can download the new root certificate from an insecure source. As it is signed with the old root key, it cannot be modified. The only possible problem is if a participant does not get the new root certificate. Most systems overlap the validity of the root keys by a few months to allow sufficient time for switching to the new root key.

There is a small implementation issue here. The new CA root certificate should probably have two signatures—one with the old root key so users can recognize the new root certificate, and one (self-certifying) signature with the

new root key to be used by new devices that are introduced after the old key expires. You can do this either by including support for multiple signatures in your certificate format, or by simply issuing two separate certificates for the same new root key.

20.2 The Life of a Key

Let's consider the lifetime of a single key. This can be the CA's root key or any other public key. A key goes through several phases in its life. Not all keys require all phases, depending on the application. As an example, we'll use Alice's public key.

Creation The first step in the life of a key is creation. Alice creates a public/private key pair and stores the private part in a secure manner.

Certification The next step is certification. Alice takes her public key to the CA or the sub-CA and has it certify her key. This is the point where the CA decides which permissions to give to Alice's public key.

Distribution Depending on the application, Alice might have to distribute her certified public key before she can use it. If, for example, Alice uses her key for signatures, each party that could potentially receive Alice's signature should have her public key first. The best way to do this is to distribute the key for a while before Alice uses it the first time. This is especially important for a new root certificate. When the CA switches to a new root key, for example, everybody should be given the chance to learn the new root certificate before being presented with a certificate signed with the new key.

Whether you need a separate distribution phase depends on your application. If you can avoid it, do so. A separate distribution phase has to be explained to the users and becomes visible in the user interface. That, in turn, creates lots of extra work, because many users won't understand what it means and will not use the system properly.

Active use The next phase is when Alice uses her key actively for transactions. This is the normal situation for a key.

Passive use After the active use phase, there must be a period of time where Alice no longer uses her key for new transactions, but everybody still accepts the key. Transactions are not instantaneous; sometimes they get delayed. A signed e-mail could very well take a day or two to reach its destination. Alice should stop using her key actively and allow a reasonable period for all pending transactions to be completed before the key expires.

Expired Finally, the key expires and it is not considered to be valid anymore.

How are the key phases defined? The most common solution is to include explicit times for each phase transition in the certificate. The certificate contains the start of the distribution phase, the start of the active use phase, the start of the inactive use phase, and the expiration time. Unfortunately, all of these times have to be presented to the user, because they affect the way the certificate works, and this is probably too complicated for ordinary users to handle.

A more flexible scheme is to have a central database that contains the phase of each key, but this introduces a whole new raft of security issues, which we'd rather not do. And if you have a CRL, it can override the chosen phase periods and expire a key immediately.

Things become even more complicated if Alice wants to use the same key in several different PKIs. In general, we think this is a bad idea, but sometimes it cannot be avoided. But extra precautions need to be taken if it cannot be avoided. Suppose Alice uses a small tamper-resistant module that she carries with her. This module contains her private keys and performs the necessary computations for a digital signature. Such modules have a limited storage capacity. Alice's certificates on her public key can be stored on the corporate intranet without size limitations, but the small module cannot store an unlimited number of private keys. In situations like this, Alice ends up using the same key for multiple PKIs. It also implies that the key lifetime schedule should be similar for all the PKIs Alice uses. This might be difficult to coordinate.

If you ever work on a system like this, make sure that a signature used in one PKI cannot be used in another PKI. You should always use a single digital signature scheme, such as the one explained in Section 12.7. The signed string of bytes should not be the same in two different PKI systems or in two different applications. The simplest solution is to include data in the string to be signed that uniquely identifies the application and the PKI.

20.3 Why Keys Wear Out

We've mentioned several times that keys have to be replaced regularly, but why is this?

In a perfect world, a key could be used for a very long time. An attacker who has no system weaknesses to work with is reduced to doing exhaustive searches. In theory, that reduces our problem to one of choosing large enough keys.

The real world isn't perfect. There are always threats to the secrecy of a key. The key must be stored somewhere, and an attacker might try to get at it. The key must also be used, and any use poses another threat. The key has

to be transported from the storage location to the point where the relevant computations are done. This will often be within a single piece of equipment, but it opens up a new avenue of attack. If the attacker can eavesdrop on the communication channel used for this transport, then she gets a copy of the key. Then there is the cryptographic operation that is done with the key. There are no useful cryptographic functions that have a full proof of security. At their core, they are all based on arguments along the lines of: ''Well, none of us has found a way to attack this function, so it looks pretty safe.''[1] And as we have already discussed, side-channels can leak information about keys.

The longer you keep a key, and the more you use it, the higher the chance an attacker might manage to get your key. If you want to limit the chance of the attacker knowing your key, you have to limit the lifetime of the key. In effect, a key wears out.

There is another reason to limit the lifetime of a key. Suppose something untoward happens and the attacker gets the key. This breaks the security of the system and causes damage of some form. (Revocation is only effective if you find out the attacker has the key; a clever attacker would try to avoid detection.) This damage lasts until the key is replaced with a new key, and even then, data previously encrypted under the old key will remain compromised. By limiting the lifetime of a single key, we limit the window of exposure to an attacker who has been successful.

There are thus two advantages to short key lives. They reduce the chance that an attacker gets a key, and they limit the damage that is done if he nevertheless succeeds.

So what is a reasonable lifetime? That depends on the situation. There is a cost to changing keys, so you don't want to change them too often. On the other hand, if you only change them once a decade, you cannot be sure that the change-to-a-new-key function will work at the end of the decade. As a general rule, a function or procedure that is rarely used or tested is more likely to fail.[2] Probably the biggest danger in having long-term keys is that the change-key function is never used, and therefore will not work well when it is needed. A key lifetime of one year is probably a reasonable maximum.

Key changes in which the user has to be involved are relatively expensive, so they should be done infrequently. Reasonable key lifetimes are from one month and upwards. Keys with shorter lifetimes will have to be managed automatically.

---

[1] What is often called a ''proof of security'' for cryptographic functions is actually not a complete proof. These proofs are generally reductions: if you can break function *A*, you can also break function *B*. They are valuable in allowing you to reduce the number of primitive operations you have to assume are secure, but they do not provide a complete proof of security.

[2] This is a generally applicable truism and is the main reason you should always test emergency procedures, such as fire drills.

20.4 Going Further

Key management is not just a cryptographic problem. It is a problem of interfacing with the real world. The specific choice of which PKI to use, along with how the PKI is configured, will depend on the specifics of the application and the environment in which it is supposed to be deployed. We have outlined the key issues to consider.

20.5 Exercises

Exercise 20.1 What fields do you think should appear in a certificate, and why?

Exercise 20.2 What are the root SSL keys hard-coded within your Web browser of choice? When were these keys created? When do they expire?

Exercise 20.3 Suppose you have deployed a PKI, and that the PKI uses certificates in a certain fixed format. You need to update your system. Your updated system needs to be backward compatible with the original version of the PKI and its certificates. But the updated system also needs certificates with extra fields. What problems could arise with this transition? What steps could you have taken when originally designing your system to best prepare for an eventual transition to a new certificate format?

Exercise 20.4 Create a self-signed certificate using the cryptography packages or libraries on your machine.

Exercise 20.5 Find a new product or system that uses a PKI. This might be the same product or system that you analyzed for Exercise 1.8. Conduct a security review of that product or system as described in Section 1.12, this time focusing on the security and privacy issues surrounding the use of the PKI.