CHAPTER

19

PKI Reality

While very useful, there are some fundamental problems with the basic idea of a PKI. Not in theory, but then, theory is something very different from practice. PKIs simply don't work in the real world the way they do in the ideal scenario discussed in Chapter 18. This is why much of the PKI hype has never matched the reality.

When talking about PKIs, our view is much broader than just e-mail and the Web. We also consider the role of PKIs in authorization and other systems.

19.1 Names

We'll start with a relatively simple problem: the concept of a name. The PKI ties Alice's public key to her name. What is a name?

Let's begin in a simple setting. In a small village, everybody knows everybody else by sight. Everybody has a name, and the name is either unique or will be made unique. If there are two Johns, they will quickly come to be called something like Big John and Little John. For each name there is one person, but one person might have several names; Big John might also be called Sheriff or Mr. Smith.

The name we are talking about here is not the name that appears on legal documents. It is the name that people use to refer to you. A name is really any kind of signifier that is used to refer to a person, or more generally, to an entity. Your ''official'' name is just one of many names, and for many people it is one that is rarely used.

As the village grows into a town, the number of people increases until you no longer know them all. Names start losing their immediate association with a person. There might only be a single J. Smith in town, but you might not know him. Names now start to lead a life of their own, divorced from the actual person. You start talking about people you have never actually met. Maybe you end up talking in the bar about the rich Mr. Smith who just moved here and who is going to sponsor the high school football team next year. Two weeks later, you find out that this is the same person who joined your baseball team two months ago, and whom you know by now as John. People still have multiple names, after all. It just isn't obvious which names belong together, and which person they refer to.

As the town grows into a city, this changes even more. Soon you will only know a very small subset of the people. What is more, names are no longer unique. It doesn't really help to know that you are looking for a John Smith if there are a hundred of them in the city. The meaning of a name starts to depend on the context. Alice might know three Johns, but at work when she talks about ''John,'' it is clear from the context that she means John who works upstairs in sales. Later at home, it might mean John the neighbor's kid. The relationship between a name and a person becomes even fuzzier.

Now consider the Internet. Over a billion people are online. What does the name ''John Smith'' mean there? Almost nothing: there are too many of them. So instead of more traditional names we use e-mail addresses. You now communicate with `jsmith533@yahoo.com`. That is certainly a unique name, but in practice it does not link to a person in the sense of someone you will ever meet. Even if you could find out information such as his address and phone number, he is just as likely to live on the other side of the world. You are never going to meet him in person unless you really set out to do so. Not surprisingly, it is not uncommon for people to take on different online personalities. And as always, each person has multiple names. Most users acquire multiple e-mail addresses after a while. (We have more than a dozen among us.) But it is extremely difficult to find out whether two e-mail addresses refer to the same person. And to make things more complicated, there are people who share an e-mail address, so that ''name'' refers to them both.

There are large organizations that try to assign names to everybody. The best-known ones are governments. Most countries require each person to have a single official name, which is then used on passports and other official documents. The name itself is not unique—there are many people with the same name—so in practice it is often extended with things like address, driver's license number, and date of birth. This still does not guarantee a unique identifier for a person, however.[1] Also, several of these identifiers can change over the course of a person's life. People change their addresses,

[1]Driver's license numbers are unique, but not everybody has one.

driver's license numbers, names, and even gender. Just about the only thing that doesn't change is the date of birth, but this is compensated for by the fact that plenty of people lie about their date of birth, in effect changing it.

Just in case you thought that each person has a single government-sanctioned official name, this isn't true, either. Some people are stateless and have no papers at all. Others have dual nationalities, with two governments each trying to establish an official name—and for various reasons, they may not agree on what the official name should be. The two governments might use different alphabets, in which case the names cannot be the same. Some countries require a name that fits the national language and will modify foreign names to a similar ''proper'' name in their own language.

To avoid confusion, many countries assign unique numbers to individuals, like the Social Security number (SSN) in the United States or the SoFi number in the Netherlands. The whole point of this number is to provide a unique and fixed name for an individual, so his actions can be tracked and linked together. To a large degree these numbering schemes are successful, but they also have their weaknesses. The link between the actual human and the assigned number is not very tight, and false numbers are used on a large scale in certain sectors of the economy. And as these numbering schemes work on a per-country basis, they do not provide global coverage, nor do the numbers themselves provide global uniqueness.

One additional aspect of names deserves mention. In Europe, there are privacy laws that restrict what kind of information an organization can store about people. For example, a supermarket is not allowed to ask for, store, or otherwise process an SSN or SoFi number for its loyalty program. This restricts the reuse of government-imposed naming schemes.

So what name should you use in a PKI? Because many people have many different names, this becomes a problem. Maybe Alice wants to have two keys, one for her business and one for her private correspondence. But she might use her maiden name for her business and her married name for her private correspondence. Things like this quickly lead to serious problems if you try to build a universal PKI. This is one of the reasons why smaller application-specific PKIs work much better than a single large one.

19.2 Authority

Who is this CA that claims authority to assign keys to names? What makes that CA authoritative with respect to these names? Who decides whether Alice is an employee who gets VPN access or a customer of the bank with restricted access?

For most of our examples, this is a question that is simple to answer. The employer knows who is an employee and who isn't; the bank knows who is

a customer. This gives us our first indication of which organization should be the CA. Unfortunately, there doesn't seem to be an authoritative source for the universal PKI. This is one of the reasons why a universal PKI cannot work.

Whenever you are planning a PKI, you have to think about who is authorized to issue the certificates. For example, it is easy for a company to be authoritative with regard to its employees. The company doesn't decide what the employee's name is, but it does know what name the employee is known by *within the company*. If ''Fred Smith'' is officially called Alfred, this does not matter. The name ''Fred Smith'' is a perfectly good name within the context of the employees of the company.

19.3 Trust

Key management is the most difficult problem in cryptography, and a PKI system is one of the best tools that we have to solve it with. But everything depends on the security of the PKI, and therefore on the trustworthiness of the CA. Think about the damage that can be done if the CA starts to forge certificates. The CA can impersonate anyone in the system, and security completely breaks down.

A universal PKI is very tempting, but trust is really the area where it fails. If you are a bank and you need to communicate with your customers, would you trust some dot-com on the other side of the world? Or even your local government bureaucracy? What is the total amount of money you could lose if the CA does something horribly wrong? How much liability is the CA willing to take on? Will your local banking regulations allow you to use a foreign CA? These are all enormous problems. Just imagine the damage that can occur if the CA's private key is published on a website.

Think of it in traditional terms. The CA is the organization that hands out the keys to the buildings. Most large office buildings have guards, and most guards are hired from an outside security service. The guards verify that the rules are being obeyed: a rather straightforward job. But deciding who gets which keys is not something that you typically outsource to another company, because it is a fundamental part of the security policy. For the same reason, the CA functionality should not be outsourced.

No organization in the world is trusted by everybody. There isn't even one that is trusted by *most* people. Therefore, there will never be a universal PKI. The logical conclusion is that we will have to use lots of small PKIs. And this is exactly the solution we suggest for our examples. The bank can be its own CA; after all, the bank trusts itself, and all the customers already trust the bank

with their money. A company can be its own CA for the VPN, and the credit card organization can also run its own CA.

An interesting observation here is that the trust relationships used by the CA are ones that already exist and are based on contractual relationships. This is always the case when you design cryptographic systems: the basic trust relationships you build on are all based on contractual relationships.

19.4 Indirect Authorization

Now we come to a big problem with the classic PKI dream. Consider authorization systems. The PKI ties keys to names, but most systems are not interested in the name of the person. The banking system wants to know which transactions to authorize. The VPN wants to know which directories to allow access to. None of these systems cares *who* the key belongs to, only *what* the keyholder is authorized to do.

To this end, most systems use some kind of *access control list*, or ACL. This is just a database of who is authorized to do what. Sometimes it is sorted by user (e.g., Bob is allowed the following things: access files in the directory `/homes/bob`, use of the office printer, access to the file server), but most systems keep the database indexed by action (e.g., charges to this account must be authorized by Bob or Betty). Often there are ways to create groups of people to make the ACLs simpler, but the basic functionality remains the same.

So now we have three different objects: a key, a name, and permission to do something. What the system wants to know is which key authorizes which action, or in other words, whether a particular key has a particular permission. The classic PKI solves this by tying keys to names and using an ACL to tie names to permissions. This is a roundabout method that introduces additional points of attack [45].

The first point of attack is the name–key certificate provided by the PKI. The second point of attack is the ACL database that ties names to permissions. The third point of attack is name confusion: with names being such fuzzy things, how do you compare whether the name in the ACL is the same as the name in the PKI certificate? And how do you avoid giving two people the same name?

If you analyze this situation, you will clearly see that the technical design has followed the naive formulation of the requirements. People think of the problem in terms of identifying the key holder and who should have access—that is how a security guard would approach the problem. Automated systems can use a much more direct approach. A door lock doesn't care who is holding the key, but allows access to anyone with the key.

19.5 Direct Authorization

A much better solution is generally to directly tie the permissions to the key, using the PKI. The certificate no longer links the key to a name; it links the key to a set of permissions [45].

All systems that use the PKI certificates can now decide directly whether to allow access or not. They just look at the certificate provided and see if the key has the appropriate permissions. It is direct and simple.

Direct authorization removes the ACL and the names from the authorization process, thereby eliminating these points of attack. Some of the problems will, of course, reappear at the point where certificates are issued. Someone must decide who is allowed to do what, and ensure that this decision is encoded in the certificates properly. The database of all these decisions becomes the equivalent of the ACL database, but this database is less easy to attack. It is easy to distribute to the people making the decisions, removing the central ACL database and its associated vulnerabilities. Decision makers can just issue the appropriate certificate to the user without further security-critical infrastructure. This also removes much of the reliance on names, because the decision makers are much further down in the hierarchy and have a much smaller set of people to deal with. They often know the users personally, or at least by sight, which helps a great deal in avoiding name confusion problems.

So can we just get rid of the names in the certificates, then?

Well, no. Though the names will not be used during normal operations, we do need to provide logging data for audits and such. Suppose the bank just processed a salary payment authorized by one of the four keys that has payment authority for that account. Three days later, the CFO calls the bank and asks why the payment was made. The bank knows the payment was authorized, but it has to provide more information to the CFO than just a few thousand random-looking bits of public-key data. This is why we still include a name in every certificate. The bank can now tell the CFO that the key used to authorize the payment belonged to ''J. Smith,'' which is enough for the CFO to figure out what happened. But the important thing here is that the names only need to be meaningful to humans. The computer never tries to figure out whether two names are the same, or which person the name belongs to. Humans are much better at dealing with the fuzzy names, whereas computers like simple and well-specified things such as sets of permissions.

19.6 Credential Systems

If you push this principle further, you get a full-fledged credential system. This is the cryptographer's super-PKI. Basically, it requires that you need a credential in the form of a signed certificate for every action you perform.

If Alice has a credential that lets her read and write a particular file, she can delegate some or all of her authority to Bob. For example, she could sign a certificate on Bob's public key that reads something like ''Key PK_{Bob} is authorized to read file *X* by delegated authority of key PK_{Alice}.'' If Bob wants to read file *X*, he has to present this certificate and a certificate proof that Alice has read access to file *X*.

A credential system can add additional features. Alice could limit the time validity of the delegation by including the validity period in the certificate. Alice might also limit Bob's ability to delegate the authority to read file *X*.[2]

In theory, a credential system is extremely powerful and flexible. In practice, they are rarely used. There are several reasons for this.

First of all, credential systems are quite complex and can impose a noticeable overhead. Your authority to access a resource might depend on a chain of half-a-dozen certificates, each of which has to be transmitted and checked.

The second problem is that credential systems invite a micromanagement of access. It is so easy to split authorities into smaller and smaller pieces that users end up spending entirely too much time deciding exactly how much authority to delegate to a colleague. This time is often wasted, but a bigger problem is the loss of the colleague's time when it turns out he doesn't have enough access to do his job. Maybe this micromanagement problem can be solved with better user education and better user interfaces, but that seems to be an open problem. Some users avoid the micromanagement problem by delegating (almost) all their authority to anyone who needs any kind of access, effectively undermining the entire security system.

The third problem is that you need to develop a credential and delegation language. The delegation messages need to be written in some sort of logical language that computers can understand. This language needs to be powerful enough to express all the desired functionality, yet simple enough to allow fast chaining of conclusions. It also has to be future-proof. Once a credential system is deployed, every program will need to include code to interpret the delegation language. Upgrading to a new version of the delegation language can be very difficult, especially since security functionality spreads into every piece of a system. Yet it is effectively impossible to design a delegation language that is general enough to satisfy all future requirements, since we never know what the future will bring. This remains an area of research.

The fourth problem with credential systems is probably insurmountable. Detailed delegation of authority is simply too complex a concept for the average user. There doesn't seem to be a way of presenting access rules to

[2]This is an often-requested feature, but we believe it may not always be a good one. Limiting Bob's ability to delegate his authority just invites him to run a proxy program so that other people can use his credential to access a resource. Such proxy programs undermine the security infrastructure and should be banned, but this is only tenable if there are no operational reasons to run a proxy. And there are always operational reasons why someone needs to delegate authority.

users in a manner they can understand. Asking users to make decisions about which authorities to delegate is bound to fail. We see that in the real world already. In some student houses it is customary for one person to go to the ATM and get cash for several people. The other students lend him their ATM card and PIN code. This is an eminently risky thing to do, yet it is done by some of the supposedly more well-educated people in our society. As consultants, we've visited many companies and sometimes had work-related reasons to have access to the local network. It is amazing how much access we got. We've had system administrators give us unrestricted access to the research data, when all we needed to do was look at a file or two. If system administrators have a hard time getting this right, ordinary users certainly will, too.

As cryptographers, we'd love the idea of a credential system if only the users were able to manage the complexity. There is undoubtedly a lot of interesting research to do on human interactions with security systems.

There is, however, one area where credentials are very useful and should be mandatory. If you use a hierarchical CA structure, the central CA signs certificates on the keys of the sub-CAs. If these certificates do not include any kind of restriction, then each sub-CA has unlimited power. This is problematic; we've just multiplied the number of places where system-critical keys are stored.

In a hierarchical CA structure, the power of a sub-CA should be limited by including restrictions in the certificate on its key. This requires a credential-like delegation language for CA operations. Exactly what type of restrictions you'd want to impose depends on the application. Just think about what type of sub-CAs you want to create and how their power should be limited.

19.7 The Modified Dream

Let's summarize all the criticism of PKIs we've presented so far and present a modified dream. This is a more realistic representation of what a PKI should be.

First of all, each application has its own PKI with its own CA. The world consists of a large number of small PKIs. Each user is a member of many different PKIs at the same time.

The user must use different keys for each PKI, as he cannot use the same key in different systems without careful coordination in the design of the two systems. The user's key store will therefore contain dozens of keys, requiring tens of kilobytes of storage space.

The PKI's main purpose is to tie a credential to the key. The bank's PKI ties Alice's key to the credential that allows access to Alice's account. Or the company's PKI ties Alice's key to a credential that allows access to the VPN. Significant changes to a user's credentials require a new certificate to be issued. Certificates still contain the user's name, but this is mainly for management and auditing purposes.

This modified dream is far more realistic. It is also more powerful, more flexible, and more secure than the original dream. It is very tempting to believe that this modified dream will solve your key management problems. But in the next section, we will encounter the hardest problem of all—one that will never be solved fully and will always require compromises.

19.8 Revocation

The hardest problem to solve in a PKI is *revocation*. Sometimes a certificate has to be withdrawn. Maybe Bob's computer was hacked and his private key was compromised. Maybe Alice was transferred to a different department or even fired from the company. You can think of all kinds of situations where you want to revoke a certificate.

The problem is that a certificate is just a bunch of bits. These bits have been used in many places and are stored in many places. You can't make the world forget the certificate, however hard you try. Bruce lost a PGP key more than a decade ago; he still gets e-mail encrypted with the corresponding certificate.[3] Even trying to make the world forget the certificate is unrealistic. If a thief breaks into Bob's computer and steals his private key, you can be certain he also made a copy of the certificate on the corresponding public key.

Each system has its own requirements, but in general, revocation requirements differ in four variables:

- *Speed of revocation*. What is the maximum amount of time allowed between the revocation command and the last use of the certificate?
- *Reliability of revocation*. Is it acceptable that under some circumstances revocation isn't fully effective? What residual risk is acceptable?
- *Number of revocations*. How many revocations should the revocation system handle at a time?
- *Connectivity*. Is the party checking the certificates online at the time of certificate verification?

There are three workable solutions to the revocation problem: revocation lists, fast expiration, and online certificate verification.

19.8.1 Revocation List

A *certificate revocation list*, or CRL, is a database that contains a list of revoked certificates. Everybody who wants to verify a certificate must check the CRL database to see if the certificate has been revoked.

[3]PGP has its own strange PKI-like structure called the *web of trust*. Those interested in PGP's web of trust should read [130].

A central CRL database has attractive properties. Revocation is almost instantaneous. Once a certificate has been added to the CRL, no further transactions will be authorized. Revocation is also very reliable, and there is no direct upper limit on how many certificates can be revoked.

The central CRL database also has significant disadvantages. Everybody must be online all the time to be able to check the CRL database. The CRL database also introduces a single point of failure: if it is not available, no actions can be performed. If you try to solve this by authorizing parties to proceed whenever the CRL is unavailable, attackers will use denial-of-service attacks to disable the CRL database and destroy the revocation capability of the system.

An alternative is to have a distributed CRL database. You could make a redundant mirrored database using a dozen servers spread out over the world and hope it is reliable enough. But such redundant databases are expensive to build and maintain and are normally not an option. Don't forget, people rarely want to spend money on security.

Some systems simply send copies of the entire CRL database to every device in the system. The U.S. military STU-III encrypted telephone works in this manner. This is similar to the little booklets of stolen credit card numbers that used to be sent to each merchant. It is relatively easy to do. You can just let every device download the updated CRL from a Web server every half hour or so, at the cost of increasing the revocation time. However, this solution restricts the size of the CRL database. Most of the time you can't afford to copy hundreds of thousands of CRL entries to every device in the system. We've seen systems where the requirements state that every device must be capable of storing a list of 50 CRL entries, which can be problematic.

In our experience, CRL systems are expensive to implement and maintain. They require their own infrastructure, management, communication paths, and so on. A considerable amount of extra functionality is required just to handle the comparatively rarely used functionality of revocation.

19.8.2 Fast Expiration

Instead of revocation lists, you can use *fast expiration*. This makes use of the already existing expiration mechanism. The CA simply issues certificates with a very short expiration time, ranging anywhere from 10 minutes to 24 hours. Each time Alice wants to use her certificate, she gets a new one from the CA. She can then use it for as long as it remains valid. The exact expiration speed can be tuned to the requirements of the application, but a certificate validity period of less than 10 minutes does not seem to be very practical.

The major advantage of this scheme is that it uses the already available certificate issuing mechanism. No separate CRL is required, which significantly reduces the overall system complexity. All you need to do to revoke a

permission is inform the CA of the new access rules. Of course, everybody still needs to be online all the time to get the certificates reissued.

Simplicity is one of our main design criteria, so we prefer fast expiration to a CRL database. Whether fast expiration is possible depends mostly on whether the application demands instantaneous revocation, or whether a delay is acceptable.

19.8.3 Online Certificate Verification

Another alternative is *online certificate verification*. This approach, which is embodied in the Online Certificate Status Protocol (OCSP), has seen a lot of headway in some domains, such as Web browsers.

To verify a certificate, Alice queries a trusted party—such as the CA or a delegated party—with the serial number of the certificate in question. The trusted party looks up the status of the certificate in its own database, and then sends a signed response to Alice. Alice knows the trusted party's public key and can verify the signature on the response. If the trusted party says the certificate is valid, Alice knows that the certificate has not been revoked.

Online certificate verification has a number of attractive properties. As with CRLs, revocation is almost instantaneous. Revocation is also very reliable. Online certificate verification also shares some disadvantages with CRLs. Alice must be online to verify a certificate, and the trusted party becomes a point of failure.

In general, we prefer online certificate verification to CRLs. Online certificate verification avoids the problem of massively distributing the CRLs and avoids the need to parse and verify the CRLs on the client. The design of online certificate verification protocols can therefore be made cleaner, simpler, and more scalable than CRLs.

In most situations online certificate verification is inferior to fast expiration, however. With online certificate verification, you can't trust the key without a trusted party's signature. If you view that signature as a new certificate on the key, you have a fast-expiration system with very short expiration times. The disadvantage of online certificate verification is that every verifier has to query the trusted party, whereas for fast expiration the prover can use the same CA signature for many verifiers.

19.8.4 Revocation Is Required

Because revocation can be hard to implement, it becomes very tempting not to implement it at all. Some PKI proposals make no mention of revocation. Others list the CRL as a future extension possibility. In reality, a PKI without some form of revocation is pretty useless. Real-life circumstances mean that keys *do* get compromised, and access has to be revoked. Operating a PKI without a

working revocation system is somewhat like operating a ship without a bilge pump. In theory, the ship should be watertight and it shouldn't need a bilge pump. In practice, there is always water collecting in the bottom of the ship, and if you don't get rid of it, the ship eventually sinks.

19.9 So What Is a PKI Good For?

At the very beginning of our PKI discussion, we stated that the purpose of having a PKI is to allow Alice and Bob to generate a shared secret key, which they use to create a secure channel, which they in turn use to communicate securely with each other. Alice wants to authenticate Bob (and vice versa) without talking to a third party. The PKI is supposed to make this possible.

But it doesn't.

There is *no* revocation system that works entirely offline. It is easy to see why. If neither Alice nor Bob contacts any outside party, neither of them can ever be informed that one of their keys has been revoked. So the revocation checks force them to go online. Our revocation solutions require online connections.

But if we are online, we don't need a big complex PKI. We can achieve our desired level of security by simply setting up a central key server, such as those described in Chapter 17.

Let's compare the advantages of a PKI over a key server system:

- A key server requires everybody to be online in real time. If you can't reach the key server, you can't do anything at all. There is no way Alice and Bob can recognize each other. A PKI gives you some advantages. If you use expiration for revocation, you only need to contact the central server once in a while; for applications that use certificates with validity periods of hours, the requirement for real-time online access and processing is significantly relaxed. This is useful for non-interactive applications like e-mail. This is also useful for certain authorization systems, or cases where communications are expensive. Even if you use a CRL database, you might have rules on how to proceed if the CRL database cannot be reached. Credit card systems have rules like this. If you can't get automatic authorization, any transaction up to a certain amount is okay. These rules would have to be based on a risk analysis, including the risk of a denial-of-service attack on the CRL system, but at least you get the option of proceeding; the key server solution provides no alternatives.

- The key server is a single point of failure. Distributing the key server is difficult, since it contains all the keys in the system. You really don't want to start spreading your secret keys throughout the world. The

CRL database, in contrast, is much less security-critical and is easier to distribute. The fast-expiration solution makes the CA a point of failure. But large systems almost always have a hierarchical CA, which means that the CA is already distributed, and failures affect only a small part of the system.

- In theory, a PKI should provide you with nonrepudiation. Once Alice has signed a message with her key, she should not be able to later deny that she signed the message. A key server system can never provide this; the central server has access to the same key that Alice uses and can therefore forge an arbitrary message to make it look as if Alice sent it. In real life, nonrepudiation doesn't work because people cannot store their secret keys sufficiently well. If Alice wants to deny that she signed a message, she is simply going to claim that a virus infected her machine and stole her private key.
- The most important key of a PKI is the CA root key. This key does not have to be stored in a computer that is online. Rather, it can be stored securely and only loaded into an offline computer when needed. The root key is only used to sign the certificates of the sub-CAs, and this is done only rarely. In contrast, the key server system has the master key material in an online computer. Computers that are offline are much harder to attack than those that are online, so this makes a PKI potentially more secure.

So there are a few advantages to PKIs. They are nice to have, but none of them gives you a really critical advantage in some environments. These advantages only come at a stiff price. A PKI is much more complex than a key server system, and the public-key computations require a lot more computational power.

19.10 What to Choose

So how should you set up your key management system? Should you use a key server-type scheme or a PKI-type scheme? As always, this depends on your exact requirements, the size of your system, your target application, and so on.

For small systems, the extra complexity of a PKI is in general not warranted. We think it is easier to use the key server approach. This is mainly because the advantages of a PKI over the key server approach are more relevant for large installations than for small ones.

For large systems, the additional flexibility of a PKI is still attractive. A PKI can be a more distributed system. Credential-style extensions allow the central

CA to limit the authority of the sub-CAs. This in turn makes it easy to set up small sub-CAs that cover a particular area of operations. As the sub-CA is limited in the certificates it can issue by the certificate on its own key, the sub-CA cannot pose a risk to the system as a whole. For large systems, such flexibility and risk limitation are important.

If you are building a large system, we would advise you to look very seriously at a PKI solution, but do compare it to a key server solution. You'll have to see if the PKI advantages outweigh its extra cost and complexity. One problem might be that you really want to use credential-style limitations for your sub-CAs. To do this, you must be able to express the limitations in a logical framework. There is no generic framework in which this can be done, so this ends up being a customer-specific part of the design. It probably also means that you cannot use an off-the-shelf product for your PKI, as they are unlikely to have appropriate certificate restriction language.

19.11 Exercises

Exercise 19.1 What bad things could happen if Alice uses the same keys with multiple PKIs?

Exercise 19.2 Suppose a system employs devices that are each capable of storing a list of 50 CRL entries. How can this design decision lead to security problems?

Exercise 19.3 Suppose a system uses a PKI with a CRL. A device in that system is asked to verify a certificate but cannot access the CRL database because of a denial-of-service attack. What are the possible courses of action for the device, and what are the advantages and disadvantages of each course of action?

Exercise 19.4 Compare and contrast the advantages and disadvantages of PKIs and key servers. Describe one example application for which you would use a PKI. Describe one example application for which you would use a key server. Justify each of your decisions.

Exercise 19.5 Compare and contrast the advantages and disadvantages of CRLs, fast revocation, and online certificate verification. Describe one example application for which you would use a CRL. Describe one example application for which you would use fast revocation. Describe one example application for which you would use online certificate verification. Justify each of your decisions.