version number doesn't match. 40 The message type is unsupported. 41 The checksum didn't check. (The documentation describes this error as message stream modified, but we prefer not to place blame—there might be a perfectly innocent explanation.) code reason 330 KERBEROS V5 12.15.12 42 The message is out of order. In both encrypted and integrity-checked data, there is a sequence number. The network can certainly reorder messages, and such innocent reordering of messages should not generate an error. If the messages are being delivered with a reliable transport layer protocol, then this error would be generated because of some deliberate tampering with the message stream, which can be detected by Kerberos. If the messages are being delivered with a datagram service (like UDP), then since the sequence number is optional, it should not be used. 44 The specified version of the key is not available. 45 Bob doesn't know his key. 46 Mutual authentication failed. 47 The message direction is incorrect. In V4 this was determined based on the D BIT. The D BIT no longer exists in V5, but instead, in integrity-protected and encrypted data, there is a SENDER'S ADDRESS and a RECEIVER'S ADDRESS field. If Bob receives a message from Alice and these fields are swapped, it indicates that an intruder is trying to trick them by mirroring messages back. 48 Alternative authentication method required. For instance, Bob does not know his master key, but does have a TGT (see §12.12 Double TGT Authentication). 49 The sequence number in the message is incorrect. 50 Inappropriate type of checksum in message. Presumably, this is because Bob doesn't support that checksum type. The wording in the documentation implies that Bob is making a value judgement on Alice's choice of checksum. For instance, if Alice were to choose CRC-32 as an integrity check instead of a message digest, Bob might sneeringly send this error message. 60 Generic error. The description is in E-TEXT. 61 Some field is too long for this implementation. 62 Alice's certificate is unacceptable to the KDC (PKINIT). 63 The KDC doesn't have a certificate from an acceptable CA (PKINIT). 64 Alice's signature is invalid (PKINIT). 65 Alice's key isn't big enough to suit the KDC. 66 Alice's name doesn't match the name in her certificate. 70 The KDC can't verify Alice's certificate. 71 One of Alice's certificates is invalid. 72 One of Alice's certificates is revoked. 73 The KDC can't figure out whether one of Alice's certificates is revoked. 74 The KDC can't reach Alice's certificate revocation service. 75 Alice's name doesn't match the name in her certificate. 76 The KDC is not who Alice thinks it is. code reason 12.16 HOMEWORK 331 12.16 HOMEWORK 1. Suppose the Kerberos V5 password to key conversion function is identical to V4, but then takes the output that V4 would compute and $\oplus$s it with the realm name. This would produce a different key in each realm, as desired. What is wrong with this algorithm? (Hint: the reason it is good for your key to be different in different realms is so that if your key in one realm is known, it does not divulge your key in other realms.) 2. Consider the following variant of Kerberos. Instead of having postdated or renewable tickets, a server which notes that the start-time is older than some limit presents the ticket to the TGS and asks if it should believe the ticket. What are the trade-offs of this approach relative to the Kerberos V5 approach? 3. The philosophy behind requiring renewable tickets to be renewed before they expire is that a KDC should not need to remember blacklist information indefinitely. But does that work for postdated tickets, given that a postdated ticket can be requested with a start-time arbitrarily far into the future? 4. Design a different method of Bob authenticating Alice when Bob does not remember his own master key, which places the work on Bob instead of Alice. In other words, Alice will act as if Bob was an ordinary civilized thing that does remember its own master key, and Bob interacts appropriately with the KDC so that Alice will be unaware that Bob didn't know his own master key. 5. Suppose it was desired to use Kerberos for securing electronic mail. The obvious way of accomplishing this is for Alice, when sending a message to Bob, to obtain a ticket for Bob and include that in the email message, and encrypt and/or integrity-protect the email message using the key in the ticket. The problem with this is that then the KDC would give Alice a quantity encrypted with Bob's password-derived master key, and then Alice could do off-line password guessing. How might Kerberos be extended to email without allowing off-line password guessing? (Hint: issue human users an extra, unguessable master key for use with mail, and extend the Kerberos protocol to allow Bob to safely obtain his unguessable master key from the KDC.) 6. In the mutual authentication in the Needham/Schroeder protocol upon which Kerberos is based, the authenticator contained only an encrypted timestamp. The protocol is that Alice sends Bob the authenticator, and then Bob must decrypt the authenticator, add one to the value inside, re-encrypt it, and send it back to Alice. Why was it necessary for Bob to increment the value before re-encrypting it and sending it to Alice? Why isn't it necessary in Kerberos V5, in the AP_REP message? In Kerberos V4, it is the checksum field (which isn't 332 KERBEROS V5 12.16 really a checksum—see §11.10 Encryption for Integrity Only) that is extracted and incremented. Would it have been just as secure in V4 for Bob to send back the contents of the checksum field encrypted and not incremented? 7. In the KRB_SAFE message, there is both a timestamp and a sequence number. Presuming that both timestamp fields (TIMESTAMP and USEC) are sent, and that the application makes sure the timestamp increases on every message, does the sequence number provide any additional protection? 8. Prove that if there is a 64-bit value that works as a DES checksum, the value ⊕'d with F0F0F0F0F0F0F0F016 will also have a correct DES checksum. 9. Give the algorithm for verifying the MAC described in §14.8.1.2 des-mac. 333 13 PKI (PUBLIC KEY INFRASTRUCTURE) 13.1 INTRODUCTION In the early days of the Indian Territory, there were no such things as birth certificates. You being there was certificate enough. —Will Rogers A public key infrastructure (PKI) consists of the components necessary to securely distribute public keys. Ideally, it consists of certificates (see §9.7.2 Certification Authorities (CAs)), a repository for retrieving certificates, a method of revoking certificates, and a method of evaluating a chain of certificates from public keys that are known and trusted in advance (trust anchors) to the target name. There have been some public-key-based systems deployed that leave out components such as revocation, or even certificates. Whether such systems are worthy of being called PKIs is a matter for debate (and a fairly dull debate at that). In practice, many people do use public key technology for protecting communication and don't use a PKI. Instead, they exchange public keys in email, or download the public key from the IP address at which they assume their target is located. In theory, if an active attacker were watching when the initial exchange occurred, the attacker could change the public key in the message and act as a man-in-the-middle. But in practice, this mechanism is reasonably secure. In this chapter we assume certificate-based PKIs, and focus on the generic issues as well as the details of the standards. In subsequent chapters we discuss choices made in deployed systems, such as S/MIME, PGP, SSL, and Lotus Notes. This chapter assumes each entity knows its own private key. We address the particular problem of getting the private key to a human in §12.4 Strong Password Credentials Download Protocols. A certificate is a signed message vouching that a particular name goes with a particular public key, as in [Alice's public key is 829348]Carol. If Bob does not know Carol or Carol's key, then such a certificate will not help him gain confidence that 829348 is Alice's key. But if Bob knows Ted's key, then the chain [Carol's key is 348203]Ted → [Alice's public key is 829348]Carol 334 PKI (PUBLIC KEY INFRASTRUCTURE) 13.2 would mathematically allow Bob to verify Alice's key. However, there are several potential problems. Although Bob might trust Ted for vouching for Carol's key, should he trust Carol for vouching for Alice's key? Maybe Carol is careless and can be tricked into signing a certificate for a bogus key. Maybe she can be bribed. Maybe she is simply one of the bad guys trying to disrupt operations on the Internet by feeding in bad data. Furthermore, how does Bob know Ted's key? In this chapter we'll discuss issues such as these, as well the details of the certificate formats as designed by committee. 13.2 SOME TERMINOLOGY If Alice signs a certificate vouching for Bob's name and key, then Alice is the issuer and Bob is the subject. If Alice wants to find a path to Bob's key, then Bob's name is the target. If Alice is evaluating a chain of certificates, she is the verifier, sometimes called the relying party. Anything that has a public key is known as a principal. A trust anchor is a public key that the verifier has decided through some means is trusted to sign certificates. In a verifiable chain of certificates, the first certificate will have been signed by a trust anchor. 13.3 PKI TRUST MODELS Suppose Alice wants to send an encrypted email message to Bob. She needs to securely find out Bob's public key. The PKI trust model defines where Alice gets her trust anchors, and what paths would create a legal chain from a trust anchor to the target name ("Bob" in this example). 13.3.1 Monopoly Model In this model, the world chooses one organization, universally trusted by all companies, countries, universities, and other organizations to be the single CA for the world. The key of that one organization is embedded in all software and hardware as the PKI trust anchor. Everyone must get certificates from it. This is a wonderfully simple model, mathematically. This is the model favored by organizations hoping to be the monopolist. However, there are problems with it: • There is no one universally trusted organization. 13.3.2 PKI TRUST MODELS 335 • Given that all software and hardware would come preconfigured with the monopoly organization's key, it would be infeasible to ever change that key in case it were compromised, since that would involve reconfiguration of every piece of equipment and software. • It would be expensive and insecure to have a remote organization certify your key. How would they know it was you? How would you be able to securely send them your public key? Although transmission of the public key does not require secrecy, it requires integrity. Otherwise the CA could be tricked into certifying the public key as yours. • Once enough software and hardware was deployed so that it would be difficult for the world to switch organizations, the organization would have monopoly control, and could charge whatever it wanted for granting certificates. • The entire security of the world rests on that one organization never having an incompetent or corrupt employee who might be bribed or tricked into issuing bogus certificates or divulging the CA's private key. 13.3.2 Monopoly plus Registration Authorities (RAs) This model is just like §13.3.1 Monopoly Model except that the single CA chooses other organizations (known as RAs) to securely check identities and obtain and vouch for public keys. The RA then securely communicates with the CA, perhaps by sending signed email with the information that would go into the certificate, and the CA can then issue a certificate because it trusts the RA. This model's advantage over the §13.3.1 Monopoly Model is that it is more convenient and secure to obtain certificates, since there are more places to go to get certified. However, all the other disadvantages of the monopoly model apply. RAs can be added to any of the models we'll talk about. Some people believe that it is better for their organization to run an RA and leave the operation of the CA to an organization more expert at what it takes to be a CA. However, in practice, the CA just rubber-stamps whatever information is verified by the RAs. It is the RA that has to do the security-sensitive operations of ensuring the proper mapping of name to key. The CA might be better able to provide a tamper-proof audit trail of certificates it has signed. 13.3.3 Delegated CAs In this model the trust anchor CA can issue certificates to other CAs, vouching for their keys and vouching for their trustworthiness as CAs. Users can then obtain certificates from one of the delegated CAs instead of having to go to the trust anchor CA. 336 PKI (PUBLIC KEY INFRASTRUCTURE) 13.3.4 The difference between a delegated CA and an RA is whether Alice sees a chain of certificates from a trust anchor to Bob's name, or sees a single certificate. Assuming a monopoly trust anchor, this model has security and operational properties similar to §13.3.2 Monopoly plus Registration Authorities (RAs). Chains of certificates through delegated CAs can be incorporated into any of the models we'll discuss. 13.3.4 Oligarchy This is the model commonly used in browsers. In this model, instead of having products preconfigured with a single key, the products come configured with many trust anchors, and a certificate issued by any one of them is accepted. Usually in such a model it is possible for the user to examine and edit the list of trust anchors, adding or deleting trust anchors. It has the advantage over the monopoly models that the organizations chosen as trust anchors will be in competition with each other, so the world might be spared monopoly pricing. However it is likely to be even less secure than the monopoly model: • In the monopoly model, if the single organization ever has a corrupt or incompetent employee, the entire security of the world is at risk. In the oligarchy model, though, any of the trust anchor organizations getting compromised will put the security of the world at risk. It is of course far more likely that at least one of n organizations will wind up with a misused key when n is bigger than 1. • The trust anchor organizations are trusted by the product vendor, not by the user. Why should the vendor decide whom the user should trust? Also, how does the vendor choose which organizations to trust? You'd like to assume that there is some elaborate procedure by which the vendor evaluates the trustworthiness of the organization before adding its key to the trust anchor set. The policy is at the discretion of the vendor, and some vendors have chosen to include any organization willing to pay for the privilege of being included in the preconfigured trust anchor set. • It might be easy to trick a naive user into adding a bogus trust anchor into the set. This depends on the implementation. One could imagine an implementation that, upon seeing a certificate signed by an organization that wasn't in the set, would show the user a pop-up box saying, Warning. This was signed by an unknown CA. Would you like to accept the certificate anyway? (The user will almost certainly say OK.) Would you like to always accept this certificate without being asked in the future? (OK.) Would you like to always accept certificates from the CA that issued that certificate? (OK.) Would you like to always accept certificates from any CA? (OK.) Since you're willing to trust anyone for anything, would you like me to make random edits to the files on your hard drive without bothering you with a pop-up box? 13.3.5 PKI TRUST MODELS 337 (OK.) (You might want to see how many of these questions your browser asks, and it would be an interesting psychology exercise to see how outrageous you can be before a user stops clicking OK.) Note that if a user is sufficiently sophisticated and careful, she can ask for information about the certificate before clicking OK to accept it. She will be informed of the name of the signer, say Mother Teresa (the most trustworthy imaginable signer). But this does not necessarily mean it was really signed by Mother Teresa. It just means that whoever signed it (say SleazeInc) put the string Mother Teresa into the ISSUER NAME field. • Users will not understand the concept of trust anchors. If they have been assured that the application they are using does encryption, they will assume that it will be secure even if they're using a public workstation, perhaps in a hotel room or at an airport. Although it will always be an issue if a user can be tricked into using a public workstation with malicious code, it would be easier for the previous user of the workstation to modify the set of trust anchors and the proxy to be used (probably not a privileged operation) than to change the software. • There is no practical way for even a knowledgeable user to be able to examine the set of trust anchors and tell if someone has modified the set. Browsers today come shipped with about 80 trust anchors. You can examine them by name, but someone could delete the key of TrustworthyInc, and put in a new key claiming that it belongs to TrustworthyInc. You might even be able to look at digests of the keys, but what user will be sufficiently paranoid to have printed out all the message digests of the 80 or so trust anchors that get shipped with the application and compare them with the configured set? 13.3.5 Anarchy Model This is the model used by PGP. Each user is responsible for configuring some trust anchors, for instance, public keys of people he has met and who have handed him a business card with a PGP fingerprint (the message digest of the public key), and sent him email containing a public key with that digest. Then anyone can sign certificates for anyone else. Some organizations (for instance, MIT does this today) volunteer to keep a certificate database into which anyone can deposit certificates. To get the key of someone whose key is not in your set of trust anchors, you can search through the public database to see if you can find a path from one of your trust anchors to the name you want. This absolutely eliminates the monopoly pricing, but it is really unworkable on a large scale: • The database would get unworkably large if it were deployed on Internet scale. If every user donated, say, ten certificates, the database would consist of billions of certificates. It would be impractical to search through the database and construct paths. 338 PKI (PUBLIC KEY INFRASTRUCTURE) 13.3.6 • Assuming somehow Alice could piece together a chain from one of her trust anchors to the name Bob, how would she know whether to trust the chain? So, Carol (her trust anchor) vouches for Ted's key. Ted vouches for Gail's key. Gail vouches for Ken's key. Ken vouches for Bob's key. Are all these individuals trustworthy? As long as this model is used within a small community where all the users are trustworthy, it will work, but on the Internet scale, when there are individuals who will purposely add bogus certificates, it would be impossible to know whether to trust a path. Some people have suggested that if you can build multiple chains to the name that you can be more assured of the trustworthiness. But once someone decides to add bogus certificates, he can create arbitrary numbers of fictitious identities and arbitrary numbers of certificates signed by those entities. So sheer numbers will not be any assurance of trustworthiness. 13.3.6 Name Constraints The concept of name constraints is that the trustworthiness of a CA is not a binary value where a CA would either be completely untrusted or trusted for everything. Instead, a CA should only be trusted for certifying some subset of the users. For instance, MIT's CA, most likely managed by playful undergraduates, should be trusted for certifying name/key binding of MIT students, but not for certifying the key of, say, president@whitehouse.gov. Assuming users have hierarchical names, such as radia@alum.mit.edu, it is easy to specify a policy for trusting the MIT CA. The MIT CA should be trusted for certifying names in the namespace under mit.edu, but not names of the form foo@harvard.edu. Although I2 might be a Sun employee, you would not trust the Sun CA to certify the name radia@alum.mit.edu. But you would trust the Sun CA to certify the name radia.perlman@sun.com. The name by which you know someone determines whom you trust to certify that name. Users might have multiple names. The PKI doesn't care. Each name is a separate PKI entity. They might use the same public key, in which case someone might happen to notice that radia@alum.mit.edu and radia.perlman@sun.com are most likely the same individual because the two entities have the same public key. Or I2 might use different keys for my2 different identities. 13.3.7 Top-Down with Name Constraints This model is similar to the monopoly model in that everyone must be configured with a preordained, never changing root key, and that root CA delegates to other CAs. However, the delegated CAs are only allowed to issue certificates for their portions of the namespace. In this model it is easy to find the path to a name (just follow the namespace from the root down). But it has the other 13.3.8 PKI TRUST MODELS 339 problems of the monopoly model, in that everyone has to agree upon a root organization, and that organization and its key would be prohibitively expensive to ever replace. 13.3.8 Bottom-Up with Name Constraints This model is not deployed, although the design of Lotus Notes is close (see §21.5 Lotus Notes Security). It was originally proposed for Digital's security architecture in the late 1980s (see §21.4 DASS/SPX). We believe this model, or something close to it, will better serve the Internet because of the reasons we give at the end of this section. The philosophy of this model is that each organization can create its own PKI and then link to others. The model assumes a hierarchical namespace in which each node is represented by a CA. Not only does the parent certify the child's name, but the child certifies the parent's name. In other words, .edu would certify mit.edu, and mit.edu would certify .edu. In addition to up-links (where the child certifies the parent) and down-links (where the parent certifies the child), cross-links are allowed, where a cross-link is a link from any node to any other node where neither is an ancestor of the other. (See Figure 13-1.) The certificate by which one node creates a cross-link to another node is known as a cross-certificate. Note that with links in both directions (from child to parent and from parent to child), it is possible to navigate the namespace starting from any node. Instead of using the root as your trust anchor, you can start anywhere—the uppermost key within your own organization, or even your own key! If the trust anchor is your own key, the only thing you need to know a priori is your own key pair. If the trust anchor is something other than your key, you also need to know the trust anchor's public key. We define an ancestor of a name to be any prefix of that name (where the strings delimited by slashes are considered atomic), including the name itself. To find a path to a target, start at your trust anchor. If it is an ancestor of the target name, go down from there to the name. If not, look for a cross-certificate to an ancestor of the target. If you don't find a suitable cross-certificate, go up to the parent, look for cross-certificates to an ancestor of the target, and so forth, until you either find a suitable cross-certificate or get to the least common ancestor of the trust anchor and the target. (The least common ancestor is the node with the longest name which is a prefix of both names.) Once at an ancestor of the target, just follow down-links to the target. The rule is you follow up-links as far as necessary (until you encounter a cross-link to an ancestor of the target at or below the least common ancestor, or until you reach the least common ancestor), then you follow at most one cross-link, and then you follow down-links from there. Without cross-links, the set of CAs you must trust are all your ancestors and all the target's ancestors up to the least common ancestor. With cross-links, the set of CAs that you must trust is a subset of that. So for instance, imagine user A/B/X in Figure 13-1 wishes to find the key of user A/C/Y, and user A/B/X uses her own key as her trust anchor. So she looks in the directory under her own record (A/B/X) for cross-certificates. Since there are none, she goes up to her parent (A/B) and looks for 340 PKI (PUBLIC KEY INFRASTRUCTURE) 13.3.8 cross-certificates. Since there are none, she goes to its parent (A), and at that point she has reached the least common ancestor, so she can go down to the target name. Now suppose A/C/Y wants to find the key of B/Y/Z/C. She'd go up to her parent (A/C), and then follow the cross link to B/Y/Z, and then go down to B/Y/Z/C. But there is no path back from B/Y/Z/C to A/C/Y since the PKI does not go up to a common ancestor of those two names, and there is no cross link from an ancestor of B/Y/Z/C to an ancestor of A/C/Y. It might look as though B/Y/Z/C could go up one level to B/Y/Z, and then down to B/Y/Z/A from which there is a cross link to A/C. In order to be able to find such a path, the search rules would be very complex, since each link would have to be followed in case it led to a cross link to the target name. But a thornier issue is whether to trust any intermediary other than the CAs up to a common ancestor. If the trust rules are clear, e.g., only CAs along the name path are trusted, then it's easy to find and blame the compromised CA, and it's also easy to know what damage can be caused by a given CA's being compromised. If you trust any CA for anything it won't be secure. If you have any rule between those two extremes, the security becomes very complicated to configure. If it were important for there to be a path from B/Y/Z/C to A/C/Y, then B/Y/Z/C or one of its parents would create a cross link to A, A/C, or A/C/Y. Eventually organizations would tire of maintaining many cross-links. At that point there arises a business opportunity to provide inter-organization connectivity (which we'll call root service), but in competition with other organizations providing root service. An organization that offers root service would advertise its rates, how much liability it is willing to assume, would explain its policies and procedures for how carefully it checks information before issuing a certificate, and so forth. We like this model. It was originally proposed for Digital's security architecture in the late 1980s. With the trust anchor being the uppermost key in one's own organization, it is similar to the PKI for Lotus Notes, and the bridge CA model used for the Federal PKI. The bridge CA is simply a CA that certifies and is certified by the uppermost CA in each organization. The advantages of this model are: • It is easy to find out if a path exists. A A/B A/C A/B/X A/B/K A/C/Y B/Y/Z B/Y/Z/A B/Y/Z/C B/Y/Z/A/C Figure 13-1. Bottom-Up PKI Model 13.3.8 PKI TRUST MODELS 341 • The policy of assuming that the name by which something is known implies whom you'd trust to certify the name is something people can understand, and is sufficiently flexible and simple that it might actually work. • PKI can be deployed in any organization independently of the rest of the world. There is no reason to pay a commercial CA for certificates. There is no reason to wait for the entire world-encompassing PKI to get put into place before you can use PKI in your own organization, or between a few organizations. • Since authentication paths between users in your own organization never go outside of your own organization, security of what is presumably the most security-sensitive operation— authenticating users in your own organization—is entirely in your own hands. Compromise of any CA outside of your own organization will not allow anyone to impersonate one of your own users to your own services. • Replacing any key is reasonably easy. For instance, assume that a few companies offering root service successfully manage to acquire a large customer base. If a root service's key gets compromised, then it only affects the top CA of each of the root service's customers. Each such CA has to revoke the old certificate it issued to the root service and issue a new certificate containing the new key, and automatically all the users in the CA's subtree are using the new key in place of the old key. • No organization gets so entrenched that it can start charging monopolistic prices. Competition is always possible. • Configuration is very easy. At the very least you need to know your own key pair. With this PKI model, that is all you need to know, since all the other CAs can be reached by paths starting with your own key. Your private key might be carried on a smart card; for other methods of obtaining your private key, see §10.4 Strong Password Credentials Download Protocols. To use a key other than your own as the trust anchor, for instance the uppermost key in your organization, you will need to also know the public key of that trust anchor. How would this be deployed? Suppose an organization, say finance.east.bigorg.com, deploys PKI-based security using this model. When someone, say Joe, is hired into that organization, he visits the CA operator. This is just another step in the process of getting hired, like visiting the badge-making office. Joe obtains a public key pair, perhaps by generating it on his own machine or obtaining a smart card. Given that the CA (like the badge facility) is probably on site, it is easy for him to physically meet the CA operator and be introduced by someone the CA operator knows. It is therefore secure and convenient for him and the CA to certify each other's public keys. Then the up-link certificate can be stored in a directory so that Joe can plug his smart card into any workstation and search the directory for all the other certificates he will need. 342 PKI (PUBLIC KEY INFRASTRUCTURE) 13.3.9 Joe may have a life other than as an employee. He might get another certificate (and name and virtual identity) from his ISP for email and from his credit card company for securely ordering things. He could decide which identity to use for any particular activity. These identities might or might not use the same public key. There may be no way of knowing when two different entities in the PKI namespace happen to map to the same carbon-based life form. 13.3.9 Relative Names Relative names is another useful concept found in DASS/SPX, useful because if an entire subtree of the namespace moves, most of the certificates do not need to be reissued. This is done by having certificates carry relative names rather than absolute names. That means that instead of putting in the entire name joe.finance.east.bigorg.com, the down-link certificate (the one from parent to child) would carry the name joe. Now, in case the company reorganizes, so that finance is moved up under bigorg.com (so it is now finance.bigorg.com), only the new certificates between bigorg.com and finance.bigorg.com need to be issued. All the certificates for the subtree under finance would remain the same. With relative names, a child certificate would carry only the component which is the extension of the parent's name. A parent certificate would not carry a name at all, but instead say this is my parent. There is an interesting issue with what to put into a cross-certificate. There are two possibilities. One possibility is to put an absolute name into a cross-certificate. That way if the issuer's portion of the namespace gets moved, but the subject's portion hasn't changed, then the certificate will still be valid. The other possibility is to put in a relative name (like ../../B/C), in which case the certificate would remain valid if a branch of the namespace containing both names were moved as a whole (see Homework Problem 2). Although relative names have some attractive properties, there are some very complex issues, such as what name to put into a cross-link and how an entity learns its own name. Since nothing with relative names has been deployed, it would be an interesting area to study. SDSI and SPKI (RFC 2693 SPKI Certificate Theory) present a design that uses a form of relative names. 13.3.10 Name Constraints in Certificates The certificate format adopted by PKIX (see §13.6 PKIX and X.509) has a field called NAME CONSTRAINTS, which allows the issuer to specify what names the subject is trusted to certify. The field can contain allowed names and disallowed names. PKIX certificates can be used to build any of the models we've mentioned. To build the bottom-up model, a child or cross-certificate would specify that the subject was only allowed to certify names in the subtree below the subject's name. A parent certificate would contain the constraint any names except myself and below. 13.3.11 PKI TRUST MODELS 343 We'd still recommend mostly building the bottom-up model, but there is some amount of flexibility that the strict up* – cross once – down* algorithm might not give. For instance, an organization might have a cross-link to other-org.com, but realizing that other-org.com also keeps crosscertificates to yet-another.com and still-another.com, the name constraint in the cross-certificate might say that the subject would be trusted to certify names in the namespaces of any of otherorg.com, yet-another.com, and still-another.com. Or there might be several root organizations that all cross-certify each other, with each having certified some subset of the organizations. Since two organizations might not have been certified by the same root, it might be necessary to go up to the root, then find a path across the roots to the target's root, and then go down. This could be accomplished by having roots cross-certify each other using the name constraint trusted for all names. The further one gets from the bottom-up model, and the closer one gets to the anarchy model, the more complex it will be to search all valid paths. 13.3.11 Policies in Certificates The PKI in PEM (see Chapter 18 PEM & S/MIME) had built-in policies. The PEM PKI consisted of a single root CA which certified multiple hierarchies, each with its own policy. Some hierarchies had pre-defined (by the standards body) policies. PEM allowed future hierarchies with different published policies. If you wanted to get certified in a particular hierarchy you had to follow the policies of that hierarchy, and you could only get certified in one hierarchy—the one whose policies you followed. Policies were intended to be things like how carefully you checked identity before issuing a certificate and how often you administered drug tests to the CA operators (see §18.5 PEM Certificate Hierarchy). The PEM PKI was a failure and was never substantially deployed, in part because of its rigidity. PKIX provides certificate extensions for policies, intended to support something along the lines of what the PEM hierarchy designers envisioned. Instead of defining what the policies are, PKIX allows for putting in OIDs, which are hierarchically assigned globally unique identifiers. The meaning of these is not standardized. Anyone can obtain an OID and define it to mean anything. Policies don't have values associated with them. So, for instance, if what you want is a policy for security level, you couldn't say policy = security level, value = confidential. Instead, you'd have to choose separate OIDs for each level of security, for example an OID for top secret, a different OID for secret, and yet another for confidential. If you want a certificate chain where every link in the chain is at least secret, then the top secret links would have to specify they meet confidential, secret, and top secret policies. It is not possible for the application to say that each link in the chain can be either secret or top secret. Instead there has to be a chain that has the same OID in each certificate. To further complicate things, it is possible, assuming two organizations are using OIDs for similar enough policies that they are willing to consider them equivalent, the cross-certificate from 344 PKI (PUBLIC KEY INFRASTRUCTURE) 13.4 one organization to the other can contain mapping rules such as OID1=OID2. That means that if a chain which must contain OID1 in the first organization crosses into the other organization's PKI, all subsequent certificates in the chain must contain OID2. The way policies are processed in a chain is that the application specifies what policy OIDs, if any, it wants to see in certificates. For example, the application might specify OID1 or OID2 or OID3. A chain must have the same OID in every link. So for instance, even if the application doesn't care whether it's OID1 or 2 or 3, if the first certificate in the chain contains only OID2 and the next certificate in the chain contains only OID3, then the chain is not valid. If the first certificate contains OID1 and OID2 and the next one contains OID2 and OID3, then the chain so far is valid, but every subsequent certificate in the chain must now contain OID2, since that was the only acceptable OID that was contained in both of the first two certificates. If policy mapping happens in the middle of the chain, and OID2 is declared equivalent to OID5, then (assuming OID2 needed to be in all the remaining certificates in the chain) OID5 must appear in all the remaining certificates in the chain. These rules are somewhat arbitrary, and whether people wind up using the PKIX policies in any useful way remains to be seen. 13.4 REVOCATION If someone realizes their key has been stolen, or if someone gets fired from an organization, it is important to be able to revoke their certificate. Certificates typically have expiration dates in them, but since it is a lot of trouble to issue a certificate (especially if the CA is off-line), the validity time is typically months, too long to wait if it needs to be revoked. This is similar to what happens with credit cards. They, too, have an expiration date. They are usually issued to be good for a year or more. However, if one is stolen, it is important to be able to revoke its validity quickly. Originally, the credit card companies published books of bad credit card numbers, and distributed these books to all the merchants. Before accepting the card, the merchant would check to make sure the credit card number wasn't in the book. This mechanism is similar to a CRL (certificate revocation list) mechanism. Today the usual mechanism for credit cards is that for each transaction the merchant calls someplace that has access to a database of invalid credit card numbers (or valid credit card numbers), and the merchant is told whether the credit card is valid (and if there is sufficient credit limit for the purchase). This is similar to an OLRS (on-line revocation service) mechanism. The PKIX standard protocol for requesting revocation status of a certificate is called OCSP (on-line certificate status protocol), and is documented in RFC 2560. 13.4.1 REVOCATION 345 Why do certificates have expiration times at all? Assuming there is a method of revoking them, the only security reason to have them expire is to make the revocation mechanism more efficient, for instance by keeping the CRL of manageable size. There are two additional real-world reasons for designing certificates with expiration times: • many deployed systems don't bother with revocation at all, and depend on expiration instead • companies that want to collect revenue from issuing certificates want to be able to collect multiple times for the same certificate. True story: At one time most browsers, by default, did not check expiration date, and even today all of them allow you to choose to ignore the expiration date in a certificate. To get PKI deployed, Verisign initially issued certificates with reasonable terms (e.g., low issuing fee) and lifetimes of two years. But once safely entrenched, the terms they demanded for renewing these certificates were far less favorable. Many server administrators noticed that most browsers didn’t check the expiration date, and so didn’t bother getting new certificates. In order to be compatible with the many servers with expired certificates, browsers are very casual about expiration date (e.g., having the default be not to check it, or making it very easy for the user to agree to ignore it). 13.4.1 Revocation Mechanisms The basic idea of a CRL is that the CA periodically issues a signed list of all the revoked certificates. This list must be issued periodically, even if no certificates have been revoked since the last CRL, since otherwise an attacker could post an old CRL (from before his certificate was revoked). If a timestamped CRL is issued periodically, then the verifier can refuse to honor any certificates if it cannot find a sufficiently recent CRL. Each CRL contains a complete list of all the unexpired, revoked certificates. 13.4.1.1 Delta CRLs Delta CRLs are intended to make CRL distribution more efficient. Let’s say you want to have revocation take effect within one hour. With a CRL, that would mean that every hour the CA would have to post a new CRL, and every verifier would have to download the latest CRL. Suppose the CRL was very large, perhaps because the company just laid off 10000 people. Every hour, every verifier would have to download a huge CRL, even though very few certificates had been revoked after that layoff. A delta CRL lists changes from the last complete CRL. The latest full CRL would have to be posted (and downloaded to each verifier) along with the periodic delta CRLs. The delta CRL would say these are all the certificates that have been revoked since February 7, 10 AM, which is the most recent full CRL. The delta CRL would be very short, often containing no certificates. Issuing delta 346 PKI (PUBLIC KEY INFRASTRUCTURE) 15.4.1.2 CRLs periodically obviates the need to issue full CRLs periodically. Instead one can issue a full CRL in place of a delta CRL when the delta CRL gets sufficiently large. 13.4.1.2 First Valid Certificate This is an idea we1,2 designed for making the CRL small again after it has become too large. This scheme also allows certificates to not have a predetermined expiration time when issued. Instead, they are only marked with a serial number, which increases every time a certificate is issued (or the issue time could be used instead of a serial number). Our version of a CRL would have one additional field that is not included in X.509. The CRL would contain a FIRST VALID CERTIFICATE field. Any certificates with lower serial numbers (or issue times) are invalid. Certificates in our scheme would have no predetermined expiration time. As long as the CRL is of manageable size there is no reason to reissue any certificates. If it looks like the CRL is getting too large, the company issues a memo warning everyone with certificate serial numbers less than some number n that they’ll need new certificates by, say, a week from the date of the memo. The number n might be the next-to-be-issued certificate serial number, or it could be some earlier one. The number n is chosen so that few of the serial numbers in the current CRL are less than n. Revoked certificates with serial numbers greater than n must continue to appear in the new CRL, while valid certificates with numbers greater than n do not have to be reissued. Some time later, say two weeks after the memo is sent, the CA issues a new CRL with n in the FIRST VALID CERTIFICATE field. Affected users (those with serial numbers less than n) who ignored the memo will thenceforth not be able to access the network until they get new certificates, since their certificates are now invalid. There are cases when even with this scheme it might be reasonable to have expiration times in certificates. For example, at a university, students might be given certificates for use of the system on a per-semester basis, with a certificate that expires after the semester. Upon paying tuition for the next semester, the student is given a new certificate. But even in those cases, it may still be reasonable to combine expiration times in some certificates with our scheme, since our scheme would allow an emergency mass-revocation of certificates. 13.4.2 OLRS Schemes An OLRS (on-line revocation server) is a system that can be queried over the net about the revocation status of individual certificates. If Alice is using service Bob, then Bob is the verifier (the one making sure Alice's certificate is valid). The design most people envision is that the server Bob queries the OLRS through some authenticated communication. You might think that introducing an on-line server into a PKI eliminates an important security advantage of public keys, because you now have an on-line trusted service. But the OLRS is not as 13.4.3 REVOCATION 347 security sensitive as a CA (or KDC). The worst the OLRS can do is claim that revoked certificates are still valid, but at least the damage is limited. It does not have a vulnerable database of user secrets (like a KDC does). Its key should be different from the CA's key, so if its key is stolen, the CA's key would not be compromised. An OLRS variant is to have Alice obtain a certificate from the OLRS declaring that as of 8 AM on June 3 Alice's certificate was not revoked. Assuming Alice will be visiting many resources, this saves the OLRS the work of talking to multiple verifiers, saves the verifier the work of querying the OLRS, and saves the network from the bandwidth used by having multiple verifiers query the OLRS. Alice would present two certificates to Bob: her long-lived certificate obtained from the CA, and the certificate of non-revocation from the OLRS. Bob can decide how quickly revocation should take effect. If he wants revocation to take place within, say, one hour, then he can insist that Alice's non-revocation certificate be timestamped within the last hour. If he complains it isn't sufficiently recent, then Alice can obtain a new one. Alice can proactively refresh her certificate, knowing that most servers would want one that is, say, less than an hour old. Then the round-trip querying of the OLRS does not need to be done at the time of a transaction. Even with Bob (instead of Alice) querying the OLRS, it is possible to do caching and refreshing. Bob can keep track of the users that tend to use his resource and proactively check with the OLRS to see if any of them have been revoked. 13.4.3 Good-lists vs. Bad-lists The standards assume that the CRL will contain all the serial numbers of bad certificates, or that the OLRS would have a database of revoked certificates. This sort of scheme is known as a bad-list scheme, since it keeps track of the bad certificates. A scheme which keeps track of the good certificates is more secure, however. Suppose the CA operator is bribed to issue a certificate, using a serial number from a valid certificate, and that no audit log indicates that this bogus certificate has been issued. Nobody will know this certificate needs to be revoked, since no legitimate person knows it was ever issued. It will not be contained in the CRL. Suppose instead that the CRL contains a list of all the valid certificates (and not just serial numbers, but hashes of the certificate for each serial number). Then the bogus certificate would not be honored, because it would not appear in the list of good certificates. There are two interesting issues with good-lists: • The good-list is likely to be much larger than the bad-list, and might change more frequently, so performance might be worse than with a bad-list. 348 PKI (PUBLIC KEY INFRASTRUCTURE) 13.5 • An organization might not want to make the list of its valid certificates public. This is easily answered by having the published good-list contain only hashes of valid certificates, rather than any other identifying information. Note that usually the good-list or bad-list, especially if publicly readable, will contain only serial numbers and hashes of the certificates rather than any other identifiable information. Then the only information divulged is the number of valid certificates (in the good-list case) or invalid certificates (in the bad-list case). There is no reason to believe that the count of good certificates is more security sensitive than the count of bad certificates. The X.509 standard says it is not permitted to issue two certificates with the same serial number, and that all certificates issued must be logged. But we can't assume that a bad guy would be hindered from issuing bogus, unaudited certificates just because it would violate the specification! 13.5 DIRECTORIES AND PKI A PKI can be facilitated by a distributed hierarchical database indexed by a hierarchical name, where associated with each name is a repository of information for that name. We call this system a directory. Each name (e.g., radia.east.sun.com) represents a node in the tree which is a record that stores information about that name, such as its IP address, or the certificates it has signed, or the certificates other principals have signed certifying its key. Each record could in theory be stored on a different machine or set of machines. To go up or down (find the parent record or a child record), the directory should keep the information necessary to find the location of the parent or child record. One widely deployed directory is DNS. It uses names such as radia.east.sun.com. The Internet works because DNS is pretty much universally deployed. Another directory standard is X.500, along with languages for querying it such as LDAP (RFC 2251). The X.500 proponents tend to scoff at DNS as being merely a “lookup service” and not a directory, since they envision the main purpose of a directory to be to answer complex queries, such as find all things that have the attribute ‘hair=red’. With DNS, you start with a specific name and look up stored attributes for that name. For many applications, especially automated ones, fast lookup based on a name is the most important thing. The Internet had gotten along just fine without an X.500-type directory, and it would not work at all today without DNS, because although there are some X.500 directories deployed, there is not a globally connected X.500 directory that you can navigate to look up information about all names, as there is with DNS. DNS has captured the low end of functionality where efficiency is needed. It provides lookup by name and nothing else. For further functionality, there are web search engines that are much 13.5.1 DIRECTORIES AND PKI 349 more flexible than X.500, and these are used extensively by people searching the web. It’s not clear a directory like X.500 has a viable niche. Today, most deployed PKIs do not use directories. Here are some ways to build a PKI without a directory: • There could be a single CA for the world. Alice would keep her own certificate, and present it when she wants to authenticate herself, or present it on request when someone wants to, for instance, send her an encrypted message. • Still with a single root CA for the world, Alice can present a chain of certificates from the root CA instead of a single certificate. • With several root CAs, Alice might have a chain from one or more of the root CAs. Bob and Alice can negotiate to find a CA that Bob trusts as a root CA for which Alice has a chain of certificates. But it’s much more convenient and flexible if there is a directory. For instance, it allows Alice to encrypt a message for Bob without first communicating with Bob. 13.5.1 Store Certificates with Subject or Issuer? Assuming there is a lookup service for retrieving information associated with each hierarchical name, under which name should certificates be stored? If Carol signs a certificate for Alice’s name/key, the certificate could be stored in Carol’s record, in Alice’s record, or in both. PKIX specifies that it must be stored in the subject’s record, but it is permissible to additionally store it in the issuer’s record. This decision by PKIX indicates a bias towards a top-down model, where parents sign certifidates for children and give the certificate to the child to store. But this decision makes it difficult to implement cross-certificates or up-certificates. Since the up-certificate or cross-certificate is for the benefit of the issuer and its descendents rather than the subject (it gives the nodes in the subtree under the issuer a path to the names in the subject’s subtree), it would not always be the case that the issuer would have write access to the subject’s record. Imagine a popular name, such as the IRS (popular in that a lot of principals will want to have a secure path to its public key). Millions of Americans might sign a cross-certificate to it. But they would not have the right to store this certificate in the IRS’s record. For a CA that offers root service, and that might have many children, it is more convenient for down-certificates from the CA to be stored in the subjects’ records, because otherwise the data in the CA’s record would get too large and difficult to search through. 350 PKI (PUBLIC KEY INFRASTRUCTURE) 13.5.2 If a principal knows its key has been compromised, it should notify everyone that has certified its key. If the certificates are stored in the subjects’ records, then the subject knows everyone that has certified its key. It has to notify all the issuers of all the certificates stored in its record. But if certificates are instead stored in the issuer's record, then the subject does not necessarily know who needs to be notified. Obviously its children and its parent would need to be notified. But it does not necessarily know which principals have signed cross-certificates. There are various solutions to this: • Make it the responsibility of the issuer to check the validity of the key periodically. This might be done by checking a URL at which the subject promises to advertise key changes, or by finding in the PKI some other certificate chain to the subject, and checking with the subject if the key found is different from the one in the cross-certificate, or periodically querying the subject about its key. • Have the ability for the issuer to request that the subject notify the issuer in the case of a key compromise. This would take less storage than having the issuer store the certificate in the subject's record. And there is no reason for this information to be stored in the subject's record; rather, it can just be kept in the subject's private storage until needed. There is one other important reason for storing the certificate in the issuer's record rather than the subject's. Except for the top-down model, it makes more sense to create a path from a trust anchor to the target name, and this is difficult to do if the certificate is stored in the subject's record. This is discussed in the next section. 13.5.2 Finding Certificate Chains To securely know Alice's public key, Bob will need to find a path from one of Bob's trust anchors to Alice. This can be done by starting with Alice and working towards the trust anchors, or vice versa. PKIX shows its bias by referring to starting with Alice as building in the forward direction, while building from a trust anchor is referred to as building in reverse. Building "forward" does not work as well if name constraints or policies are used. Suppose there was a fairly rich mesh of cross-certificates, but with name constraints used as specified in §13.3.10 Name Constraints in Certificates. If chains are built from the trust anchor, the name constraint in the certificate can tell you whether it is worth following that link in the chain. If you are trying to build a chain to a/b/c/d, and the name constraint does not include that name, then there is no need to see where that certificate might lead. If, however, you attempt to build the chain from the target, the name constraint will not help you eliminate chains that don't originate with one of your trust anchors. 13.6 PKIX AND X.509 351 Likewise with policies. If creating a chain from the trust anchor, a certificate that doesn't have the correct policy can be immediately ruled out. However, if building in the other direction, even if the application insists that OID7 must appear, and the certificate contains OID9, the path must still be explored in case a certificate closer to the trust anchor maps OID7 into OID9. 13.6 PKIX AND X.509 PKIX is a profile of X.509, which means PKIX specifies which X.509 options should be supported. X.509 is a particularly awkward format for certificates, but it seems to be what the world is assuming PKIs will be based upon. Despite its awkwardness, it is not totally unusable. We1,2 both manage to implement products with it, but we hope you'll forgive us poking a little fun at it in our description below. 13.6.1 Names How easily and efficiently the PKI concepts can be implemented is influenced by the certificate formats. Somewhat astonishingly, the IETF chose to base the certificate formats on X.509. Given how simple the basic concept is—signing a name, issue date, expiration date, and public key—X.509 was a particularly inappropriate and unfortunate choice. The names were X.500 names. We are all familiar with Internet names (known as DNS names). They look like sun.com or mit.edu. We are also familiar with Internet email names, such as user@organization.com. X.500 names, on the other hand, look like C=US, O=examplecompany name, OU=research, CN=Alice, where C means country, O means organization, OU means organizational unit, and CN is common name. There are rules about what types of name components are allowed to be under what others. And the example name C=US, O=examplecompany name, OU=research, CN=Alice is supposed to be human-friendly. That's not how it's actually encoded. The actually encoding consists of OIDs (see section §13.6.2 OIDs) for each of the name component types (C, OU, etc.). Compared to OIDs, the OU= syntax is human-friendly. There is no standard for display of X.500 names—different applications display them differently. Few if any Internet applications use X.500 names. Choosing ASN.1 as the encoding dooms certificates to be inefficient in space and computationally expensive and memory intensive to parse. What happens when you try to use a certificate format for binding names to public keys, but your applications use a different form of name than what appears in the certificate? You wind up either with security flaws or the invention of awkward work-arounds. For instance, an Internet email standard, S/MIME, uses X.509 certificates. How do you reconcile an email name such as 352 PKI (PUBLIC KEY INFRASTRUCTURE) 13.6.2 radia@alum.mit.edu with an X.500 name? Older implementations mandated that the X.500 name contain a newly invented component email=radia@alum.mit.edu (displayed differently by different applications). All the X.500 name components other than email would be ignored, though they were available for the user to examine if she so chose. The standard later specified that you should put the email name into the SUBJECTALTNAME field in the certificate. SSL had the same problem since it uses X.509 certificates. URLs contain DNS names, not X.500 names. When someone visits the site www.sun.com, and the server presents a certificate that contains an X.500 name, how can this be at all useful to reassure the user Alice that the site she is talking to is really what she expects it to be? Some browser implementations ignored the name entirely, but still made sure the certificate was properly signed. So if Alice mistakenly contacted snakeoil.com instead of her broker, the site would present a certificate with an X.500 name, Alice's browser would happily do the math and decide it was properly signed, and reassure Alice that everything was secure! It certainly isn't difficult to get a certificate. A more common (and secure) work-around is to demand that the CN portion of the X.500 name be the DNS name. Eventually X.509 added the ability to have alternate names. PKIX allows end entities (nonCAs) not to have X.500 names, but CAs still must have X.500 names even if they also have a DNS name in the SUBJECTALTNAME field in the certificate. There is no widely deployed X.500 directory. There is a widely deployed directory for Internet names—DNS. DNS may be just a "lookup service" and not a "true directory" according to the X.500 proponents' definition. But frankly, when we want to be able to look up attributes of a name, such as its IP address, its certificate, cross-certificates it has signed, etc., the fancy X.500 system doesn't let us do that because there isn't a widely deployed set of X.500 directory servers with referrals for name resolution. To be fair, certificates are not posted in DNS today, either. Given that there is no way to look up certificates in directories today, strategies in deployed systems today include emailing certificates (as in S/MIME) or sending them as part of the exchange (as in SSL/TLS, and IPsec). One place where certificates are posted today is in LDAP directories serving closed user communities (e.g., within a company). 13.6.2 OIDs An OID (object identifier) is a hierarchically assigned value consisting of a sequence of numbers separated by periods, used in ASN.1. It is a way of obtaining unique numbers for things without having any central administration hand out values. In IETF, the IANA (Internet Assigned Numbers Authority) assigns numbers, and they used to periodically publish an RFC titled Assigned Numbers which listed all the numbers that were assigned for parameter values in various protocols. For instance, in the PROTOCOL field in IP, the value 6 means TCP, 50 means ESP, and 51 means AH. 13.6.3 X.509 AND PKIX CERTIFICATES 353 The final such RFC was 1700. Now the IANA publishes the numbers on their web site as http://www.iana.org/numbers.html. With ASN.1, instead of having to send a request in to IANA for a value, you can obtain your own number by going to anyone who already has an OID. For example, if someone already has the value 1.2.840.113549.1.1.2, you can ask them to give you an OID, and they might give you 1.2.840.113549.1.1.2.79. Then you can assign all the values you want provided they all have the prefix 1.2.840.113549.1.1.2.79. As you see, OIDs can get quite large. Furthermore, different organizations can give different OIDs to the same thing. In fact, even the same organization can give different OIDs to the same thing. For instance, in the PEM standard, there are two possible ways of specifying RSA. One uses the OID 2.5.8.1.1, which is defined as RSA with one parameter specifying the number of bits in the RSA modulus. The other uses OID 1.2.840.113549.1.1.1 and takes one parameter, which is null. 13.6.3 Specification of Time UNIX time, defined a decade before ASN.1 defined a format for time, was in units of seconds, took four octets to specify, and lasted until 2038. ASN.1, a decade later, came up with a format that was also in units of seconds, but took fifteen octets to specify! Furthermore, it only had a two-digit year! The PKIX people decided that the two digits could be used for specifying years between 1950 and 2049, so at least it expired in 2049 rather than 1999. There is another name form in ASN.1 called GeneralizedTime, which has a four-digit year and uses seventeen octets. PKIX mandates use of the two-digit-year format (called UTCTime—universal coordinated time) for specifying all dates through 2049, and use of GeneralizedTime for dates after 2049. 13.7 X.509 AND PKIX CERTIFICATES An X.509 certificate contains the following information: • VERSION. There are currently three versions defined, version 1 for which the code is 0, version 2 for which the code is 1, and version 3 for which the code is 2. • SERIALNUMBER. An integer that, together with the issuing CA's name, uniquely identifies this certificate. (Note it's illegal according to the spec to issue two certificates with the same serial number, but that doesn't mean someone can't misuse the CA's key and do just that, in which case it would not, of course, uniquely identify the certificate.) 354 PKI (PUBLIC KEY INFRASTRUCTURE) 13.7 • SIGNATURE. Deceptively named, this specifies the algorithm used to compute the signature on this certificate. It consists of a subfield identifying the algorithm followed by optional parameters for the algorithm. • ISSUER. The X.500 name of the issuing CA. • VALIDITY. This contains two subfields, the time the certificate becomes valid, and the last time for which it is valid. • SUBJECT. The X.500 name of the entity whose key is being certified. This field is mandatory in X.509 but PKIX manages to make it optional, while still being conformant with X.509, by saying that it is allowed to be an empty sequence, which is kind of like a null string but in ASN.1 it takes two octets to specify. In PKIX it is permitted to utilize the optional SUBJECTALTNAME extension to name things according to the way that Internet applications would want to name things (such as using a DNS name). • SUBJECTPUBLICKEYINFO. This contains two subfields, an algorithm identifer (itself containing two subfields, one identifying the algorithm and the other providing optional parameters for it), and the subject's public key. • ISSUERUNIQUEIDENTIFIER. Optional (permitted only in version 2 and version 3, but deprecated (i.e., recommended against being used) in PKIX). Uniquely identifies the issuer of this certificate. • SUBJECTUNIQUEIDENTIFIER. Optional (permitted only in version 2 and version 3, but deprecated (i.e., recommended against being used) in PKIX). Uniquely identifies the subject of this certificate. The purpose of the optional UNIQUEIDENTIFIER fields is to eliminate the possibility of confusion when a name is reused. For example, John Smith might leave an organization and then the organization might hire another John Smith, assigning the new John Smith the same X.500 name. The author of that poem is either Homer or, if not Homer, somebody else of the same name. —Aldous Huxley • ALGORITHMIDENTIFIER. This repeats the SIGNATURE field. This field is completely and utterly redundant and didn't need to be there. PKIX renamed this field SIGNATUREALGORITHM rather than removing it. • ENCRYPTED. Perhaps it would have been better to call this field signature. But that name was already taken. Anyway, this field contains the signature on all but the last of the above fields. PKIX boldly renamed it to SIGNATUREVALUE. 13.7 X.509 AND PKIX CERTIFICATES 355 • EXTENSIONS. These are only in X.509 version 3. X.509 allows arbitrary extensions, since they are defined by OID. PKIX recommends the following. Remember, these are extensions agreed upon by a large committee of people before products seriously used any of these features, so most likely a lot of them will prove not to be terribly useful: ♦ AUTHORITYKEYIDENTIFIER. This identifies the key of the CA that signed this certificate. PKIX specifies that it should be a number that, together with the CA's name, uniquely identifies the key. ♦ SUBJECTKEYIDENTIFIER. This is typically a hash of the subject's public key, but it can be anything that, together with the subject's name, uniquely identifies the key. So even a sequence number is OK. The purpose of this field is to match against the identifier that the subject will use in the AUTHORITYKEYIDENTIFIER field if the subject signs a certificate with this key. ♦ KEYUSAGE. A bit string in which each bit specifies something for which the subject is allowed to use the key. There are nine defined, for uses such as signatures, key encipherment (the usual case of encryption with a public key, where you use the key to encrypt a secret key), data encipherment (if the data were short enough that you'd want to directly encrypt with the public key), signing X.509 certificates (which gives the subject permission to be a CA), and CRL signing (which is used to indicate the key is used for signing CRLs). CRLs are signed with the same issuer name as the CA, but usually with a different key. So the CA typically signs a certificate with its own name (and different key) as subject, without permission to sign certificates but with permission to sign CRLs. ♦ PRIVATEKEYUSAGEPERIOD. This contains two timestamps (which are the four-digityear variety since there was no backward compatibility issue). Note that the VALIDITY field already specifies when the key is supposed to be valid for. Apparently someone thought it would be useful to specify the interval when the subject is allowed to be using the key separately from the interval when the subject's certificate is valid. The conceivable use of this—and it's a stretch—is that the subject is allowed to use the key for a shorter time than verifiers might want to be able to verify the signature. ♦ CERTIFICATEPOLICIES. This is a sequence of OIDs, and, optionally, qualifier fields. Each OID represents a policy, and the optional qualifier for a policy OID might be something like a text string explaining that policy. The idea the committee had in mind was that a policy in an end entity certificate (an end entity is something other than a CA) would be something like how carefully identity was checked before the certificate was issued, and a policy in a CA-to-CA certificate would be which policies the issuing CA trusts the subject CA to assert. 356 PKI (PUBLIC KEY INFRASTRUCTURE) 13.7 ♦ POLICYMAPPINGS. This is a sequence of pairs of OIDs, mapping from a policy in the issuer's domain to a policy in the subject's domain, assuming that the policies are similar but just defined by different OIDs. ♦ SUBJECTALTNAME. This is a sequence of names. This is the way to actually use the names that Internet applications might use, such as DNS names. ♦ ISSUERALTNAME. This is encoded like SUBJECTALTNAME. ♦ SUBJECTDIRECTORYATTRIBUTES. This allows specifying attributes, such as date of birth or security clearance, of the subject. ♦ BASICCONSTRAINTS. This gives permission to the subject to issue more certificates. There are two constraints listed. One is a flag indicating whether the subject is allowed to be a CA (duplicating the KEYUSAGE flag that indicates the same thing), and the other indicates the length of chain allowed following the subject (where 0 means one more certificate is allowed in the chain). ♦ NAMECONSTRAINTS. This indicates the names for which the subject is trusted to issue certificates. Permitted as well as excluded subtrees can be specified. ♦ POLICYCONSTRAINTS. This extension allows the issuer to specify that, after n more certificates in the chain, policy mapping is no longer permitted. It also allows the issuer to specify that policy OIDs must appear in subsequent certificates, even if the application verifying the chain doesn't care. One defined policy is any policy, which means the CA doesn't care about policies. But if a previous CA wants all subsequent CAs to not only put in policies, but to not weasel out of it by using the special any policy, then it can use the INHIBITANYPOLICY extension (see below). ♦ EXTENDEDKEYUSAGE. These are additional key usages, defined by an OID, to make it easy to define new key usages. A few are defined in PKIX, with usages that are consistent with some of the usages in the regular KEYUSAGE field. For instance, one of the defined extended key usages is for timestamping, which PKIX says is consistent with KEYUSAGE of digital signature and/or nonrepudiation. ♦ CRLDISTRIBUTIONPOINTS. This describes how to find the CRL, and if the CRL issuer is not the CA, who the CRL issuer is. ♦ INHIBITANYPOLICY. This specifies that the subject (or a CA at a specified distance down the chain) is not allowed to use any policy in its POLICYCONSTRAINT field. ♦ FRESHESTCRL. Describes how to obtain delta CRLs. ♦ AUTHORITYINFOACCESS. Describes how to find information about the issuer of this certificate. 13.7.1 AUTHORIZATION FUTURES 357 ♦ SUBJECTINFOACCESS. Describes how to find information about the subject of this certificate. For instance, if the subject is a CA, it might specify how to find the certificates issued by that CA. 13.7.1 X.509 and PKIX CRLs An X.509/PKIX CRL contains the following information: • SIGNATURE. Identical to the SIGNATURE field in certificates, this specifies the algorithm used to compute the signature on this CRL. • ISSUER. Identical to the ISSUER field in certificates, this is the X.500 name of the issuing CA. • THISUPDATE. This contains the time the CRL was issued. • NEXTUPDATE. Optional. This contains the time the next CRL is expected to be issued. A reasonable policy is to treat as suspect any certificate issued by a CA whose current CRL has a NEXTUPDATE time in the past. The following three fields repeat together, once for each revoked certificate: • USERCERTIFICATE. This contains the serial number of the revoked certificate. • REVOCATIONDATE. This contains the time the certificate was revoked. • CRLENTRYEXTENSIONS. This contains various optional information such as a reason code for why the certificate was revoked. • CRLEXTENSIONS. This contains various optional information such as authority key identifier, issuer alternative name, the CRL number, the delta CRL indicator (used in delta CRLs to specify the base CRL number), and a flag indicating that the CRL only contains CA certificates or only contains end entity certificates. • ALGORITHMIDENTIFIER. As for certificates, this repeats the SIGNATURE field. • ENCRYPTED. This field contains the signature on all but the last of the above fields. 13.8 AUTHORIZATION FUTURES The two most important problems in network security are Who are you? and Should you be doing that? Authorization is the Should you be doing that? part of network security. In the remainder of 358 PKI (PUBLIC KEY INFRASTRUCTURE) 15.8.1 this chapter we discuss how authorization could be done using PKI, although no deployed PKIs today do authorization as described here. 13.8.1 ACL (Access Control List) Typically the way a server decides whether a user should have access to a resource is by first authenticating the user, and then consulting a database associated with the resource that indicates who is allowed to do what with that resource. For instance, the database associated with a file might say that Alice can read it and Bob and Carol can both read and write it. This database is often referred to as an ACL (access control list). 13.8.2 Central Administration/Capabilities Another model of authorization is that instead of listing, with each resource, the set of authorized users and their rights (e.g., read, write, execute), you would have a database that listed, for each user, everything she was allowed to do. If everything were a single application, then the ACL model and the central administration model would be basically the same, since in both cases there would be a database that listed all the authorized users and what rights each had. But in a world in which there are many resources, not all under control of the same organization, it would be difficult to have a central database listing what each user was allowed to do, and it would have scaling problems if there were many resources each user was allowed to access, especially if resources were created and deleted at a high rate. There might be a suite of applications all accessed through a single portal. For instance, you might log into the “human resources” suite of applications that would allow you to select whether you wanted to submit an expense report, record vacation time, or choose health care options. In this case you could consider the entire suite of applications as a single application with a common ACL, and even if rights for the suite are centrally administered, as long as the suite is considered as one application, it would be equivalent to the ACL model. However, it might become burdensome to have all the application-specific rights centrally administered. When the maintainer of the expense reporting application decided to add a new frill, say something that allows the user to preauthorize business class airfare, it might be easier to have an application-specific ACL that listed preauthorize business class airfare as one of the rights rather than adding it into the suite’s ACL. Some people worry that ACLs don’t scale well if there are many users allowed access to each resource. But the concept of groups answers that concern. 13.8.3 AUTHORIZATION FUTURES 359 13.8.3 Groups If there is a file that should be accessible to, say, any Sun employee, it would be tedious to type in all the names of the authorized individuals, especially if there were more than one resource with the same authorizations. The concept of a group was invented to make ACLs more scalable. It is possible to include a group name on an ACL, which means that any member of the group is allowed access to the resource. Traditionally a server that protected a resource with a group named on the ACL needed to know all the members of the group. But it is useful to allow more flexible group mechanisms in order to support cross-organizational groups where no one server is allowed to know all the members, or anonymous groups, where someone can prove membership in the group without having to divulge their identity. Traditionally groups were centrally administered, so it was easy to know all the groups to which a user belonged, and the user would not belong to many groups. But in many situations it is useful for any user to be able to create a group (such as Alice’s friends, or students who have already turned in their exams in my course), and have anyone else be able to name such a group on an ACL. Scaling up this simple concept of users, groups, and ACLs to a distributed environment has not been solved in practice. This section describes how it might be done. There exist secret-keybased systems (DCE, Windows NT, and Windows 2000/Kerberos) that distribute group membership information from a central server. 13.8.3.1 Cross-Organizational and Nested Groups An ACL should be able to contain any boolean combination of groups and individuals. Likewise, group membership should be any boolean combination of groups and individuals, e.g., the members of Alliance-executives might be CompanyA-execs, CompanyB-execs, and John Smith. Each of the groups Alliance-executives, CompanyA-execs, and CompanyB-execs is likely to be managed by a different organization and the membership is likely to be stored on different servers. How, then, can a server that protects a resource that has the group Alliance-executives on the ACL know whether to allow Alice access? If she’s not explicitly listed on the ACL, she might be a member of one of the groups on the ACL. But the server does not necessarily know all the members of the group. Let’s assume that the group name can be looked up in a directory to find out information such as its network address and its public key. • The server could periodically find every group on any ACL on any resource it protects, and attempt to collect the complete membership. This means looking up all the members of all subgroups, and subgroups of subgroups. This has scaling problems (the group memberships might be very large), performance problems (there might be a lot of traffic with servers querying group membership servers for membership lists), and cache staleness problems. How 360 PKI (PUBLIC KEY INFRASTRUCTURE) 13.8.4 often would this be done? Once a day is a lot of traffic, but also a lot of time to elapse for Alice’s group membership to take effect, and for revocations to take effect. • The server could ask the on-line group server whether Alice is a member of the group at the time Alice requests access to a resource on which a group appears. This could also be a performance nightmare with many queries, especially in the case of unauthorized users. At the least, once Alice is discovered to either belong or not belong, the server should cache this information. But again, if the cache is held for a long time it means that membership can take a long time to take effect, and revocation can also takes a long time to take effect. • All groups to which Alice belongs could be added into her Kerberos ticket. This implies that the KDC or some central authorization service knows all the groups she is in. This makes it difficult to support cross-organizational groups, where no one entity knows all the groups a user is in, and it can have scaling problems as well if a user is in many groups. • Groups to which Alice belongs could be listed in Alice's certificate. This also has scaling problems if she is in many groups. It also implies that the CA knows all the groups Alice belongs to, and requires reissuance of the certificate any time Alice joins or leaves a group. • Alice might be given a set of group membership certificates for each group to which she belongs. She could present them all whenever attempting to access a resource, or the server could request certificates for relevant groups. • The server could tell Alice in which groups she should prove membership to gain access to the resource. Then Alice, if she has membership certificates for those groups, could send the certificates to the server, or obtain group membership certificates as needed. This is an attractive solution for many reasons. In many situations it is better to have the clients do the work than the servers, because of denial-of-service attacks on the servers. Also, a single interaction with the group membership server would allow Alice to use that certificate on many servers. Further, Alice's workstation can keep track of which group memberships she has recently needed, and proactively refresh credentials. This frees servers from checking revocation status on certificates. Instead of checking for revocation, they can insist that the group membership certificate Alice presents is reasonably fresh (say less than three hours old). Each server can have its own policy for how fresh group membership certificates must be, and refuse group membership certificates staler than that. 13.8.4 Roles The term role is used in many different ways. The most common concept is that Alice can be logged in as role Alice–ordinary user or Alice–system administrator, and she gets different privi- 13.8.4 AUTHORIZATION FUTURES 361 leges depending on which role she's in. Authorization based on roles is referred to as RBAC (role based access control), and as with the term role, the term RBAC means different things to different people. Some advocates of the roles concept claim that the purpose of roles is to allow central administration of rights, instead of having an ACL for each resource. They claim such a system (which lists all the privileges that go along with a role) will scale better than an ACL model, but if centralized administration really were easier, then wouldn't the same argument apply to individuals and groups? Usually people think of a role as something that needs to be consciously invoked by a user, often requiring additional authentication such as typing a different password. In contrast, with groups it is assumed that all members are automatically given all rights of the group as long as they are members. Users may or may not be allowed to simultaneously act in multiple roles, and perhaps multiple users may or may not be allowed to simultaneously act in a particular role (like President of the United States). Some things people would like to see roles solve: • When a user is acting in a particular role, the application presents a different user interface. For instance, when a user is acting as manager, the expense reporting utility might present commands for approving expense reports, whereas when the user is acting as employee, the application might present commands for reporting expenses. • Having roles enables a user to be granted a subset of all the permissions they might have. This makes it less likely that a typo will cause a user to inadvertently do an undesirable privileged operation, because they'd only invoke the privileged role briefly, and only when necessary to do a specific action. • Allowing a user to be able to run with a subset of her rights (not invoking her most privileged role except when necessary) gives some protection from malicious code. While running untrusted code, the user would be careful to run in an unprivileged role. • Sometimes there are complex policies, such as that you are allowed to read either file A or file B but not both. Somehow, proponents of roles claim roles will solve this problem. This sort of policy is called a Chinese wall. On a single machine, some of these concepts of roles can be implemented straightforwardly. But what about in a distributed environment? Most of the functionality that people envision for roles can be done with groups. But there are three concepts: individuals, groups, and roles. What might be the difference between a role and a group? A role has to be explicitly invoked, and perhaps with additional authentication. So in that case, what is the difference between a role and an individual? Why not just consider administrator and user as different entities? The reason is that for auditing purposes it is useful to know which user was acting in the administrator role when a particular action was taken. 362 PKI (PUBLIC KEY INFRASTRUCTURE) 13.8.5 Making fancy policies work in a distributed fashion is at best a subject of research today. If you want to ensure that only one user is acting in a particular role at any time, or that a user must not be allowed to see both files A and B, a conceivable method for implementing this is to have a central service (which might for robustness or performance be implemented on multiple machines, coordinating amongst themselves) which keeps track of who has done what and grants permissions for actions. This wouldn't be a central service for the entire inter-organizational internet, but rather a service for a suite of applications. The user logs into the central server, so it can keep track of what role(s) the user has at the moment and what actions the user has taken. 13.8.5 Anonymous Groups If a user can prove she is a member of a group which is authorized access to the resource, it is not necessarily the case that she needs to divulge and prove her identity. In many cases it will be necessary, for auditing purposes. But in some cases, it might be desirable to anonymously prove group membership. This can be accomplished by having Alice authenticate herself to the group membership server, provide it with a public key P (different from Alice's long-term key), and have the group membership server issue a certificate stating that the holder of the private key associated with P is a member of the group. In order to not allow someone to correlate uses of the public key to know that the same user did both actions, a user might want to have a lot of group membership certificates for the anonymous group, each with a different key. If it is desired that even the group membership server should not know which key is associated with which member, then the group membership server could do a blind signature, a concept invented by David Chaum in which Bob signs something without knowing what he's signing! It is rather surprising that such a protocol exists, that it would be useful for anything, and that anyone would have thought of it! But assuming you'd want to be able to use your privileges as a member of the group without anyone being able to know which individual you were, this feature would be useful. With blind signatures, Bob does not know which keys belong to which members, and so cannot divulge this information. A blind signature is easy to understand. Assume the signature algorithm is RSA, and that Bob's public key is ⟨e,n⟩. If Alice wants a particular certificate c signed by Bob, then she picks a random number R, and raises R to e mod n, and multiplies c by the result. So she gets c(Re mod n). Bob can't distinguish this from a random number. He signs the result, meaning that he raises it to his private exponent d. So he computes cd (Red) mod n. Red is just R. So Alice divides what she gets from Bob by R and her certificate is now validly signed by Bob. Note that this only obscures what Bob has signed if Bob signs lots of things with that key. And note that this is only secure if Bob has a different key for each kind of assertion he signs. 13.9 HOMEWORK 363 13.9 HOMEWORK 1. Referring to Figure 13-1, which CAs must B/Y/Z/A/C trust in order to find a path to A/C/Y? 2. In a relative names PKI as described in §13.3.9 Relative