Trudy does discover JAlice, she can interpret the messages she's recorded, discover JAB, and, using the ticket 242 SECURITY HANDSHAKE PITFALLS 9.4.3 KBob{JAB,"Alice"}, she can convince Bob that Alice will be talking to him using JAB. The fact that the KDC knew Alice's key changed is irrelevant; the KDC doesn't participate in this new authentication. A paper [DENN81] pointed out this weakness, and in [NEED87], a suggested fix was proposed by adding two additional messages. Alice requests a nonce NB from Bob, which Alice will send to the KDC, and the KDC will package NB in the ticket to Bob. This will reassure Bob that whoever is talking to him has talked to the KDC since Bob generated NB. Once Alice changes her key, Trudy will not be able to talk to the KDC using Alice's old key, and any recorded messages from the KDC using an old key of Alice's will also not be useful. The proposed protocol added 2 messages, and therefore uses 7 messages. The protocol can, however, be reduced to 6 messages (see Homework Problem 9). 9.4.3 Otway-Rees In [OTWA87], an improved authentication protocol is given. It also solves the ticket invalidation problem, and it does mutual authentication in 5 messages. Otway and Rees mention that they were inspired by a remark from Needham that the suspicious party should always generate a challenge. Let's look at their protocol (Protocol 9-20). Alice generates two nonces. One, NC, is sent in the clear to Bob. In addition NC is included (along with Alice I want to talk to you Bob 1 KBob{NB} 2 N1, Alice wants Bob, KBob{NB} KDC 3 invents key KAB extracts NB KAlice{N1,"Bob",KAB,ticket to Bob} where ticket to Bob = KBob{KAB,"Alice",NB} 4 ticket, KAB{N2} 5 KAB{N2−1,N3} 6 KAB{N3−1} 7 Protocol 9-19. Expanded Needham-Schroeder 9.4.3 MEDIATED AUTHENTICATION (WITH KDC) 243 the other nonce, NA) in a message encrypted by Alice that Bob cannot interpret. It just looks like a pile of bits, and he will simply forward KAlice{NA,NC,"Alice","Bob"} to the KDC. In message 2, Bob forwards the encrypted message Alice sent, along with a message Bob encrypts which includes his own nonce NB, the nonce NC that Alice sent in the clear, "Alice", and "Bob", to the KDC. The KDC checks to make sure that the common nonce NC is the same in both encrypted messages. If not, the KDC will reject the message. The fact that they are the same proves that Bob is really Bob, since only someone knowing KBob can encrypt NC inside a message. In message 3, the KDC gives Bob a message to forward to Alice. That message, when forwarded to Alice, reassures Alice that both the KDC and Bob are legitimate. (Alice knows the KDC is legitimate because it encrypts NA with key KAlice. Alice knows Bob is legitimate because the KDC would not have continued the protocol if it hadn't already verified that Bob was legit.) The KDC also tells Bob the common key, and reassures Bob that it is really the KDC by putting Bob's nonce NB inside the message to Bob. Message 4 is merely having Bob forward KAlice{NA,KAB} to Alice. In message 5, Alice proves her identity to Bob by showing that she knows KAB. Bob does not need to prove his identity explicitly to Alice because Alice will assume that the KDC authenticated Bob. In fact, this protocol could be simplified by getting rid of one of Alice's nonces (see Homework Problem 4). If we want to be really paranoid, then for subtle reasons it is necessary in the Otway-Rees protocol that NC be not just a nonce but also unpredictable. Otherwise Trudy, in a rather involved scenario, could impersonate Bob to Alice. This is the way it would work: Suppose Alice is using a sequence number for NC. Trudy watches and sees that Alice is currently using, say, 007 for NC. So Trudy sends a message to Bob claiming to be Alice. She sends 008, "Alice", "Bob", garbage. Bob can't tell that the fourth element in the message is garbage. He forwards garbage on, along with his Alice NC, "Alice", "Bob", KAlice{NA,NC,"Alice","Bob"} Bob 1 KDC KAlice{NA,NC,"Alice","Bob"}, KBob{NB,NC,"Alice","Bob"} 2 invents key KAB extracts NB NC, KAlice{NA,KAB}, KBob{NB,KAB} 3 KAlice{NA,KAB} 4 KAB{anything recognizable} 5 Protocol 9-20. Otway-Rees 244 SECURITY HANDSHAKE PITFALLS 9.5 own message KBob{NB,008,"Alice","Bob"}. Trudy records that message. The KDC will reject Bob's message, since the garbage won't decrypt properly. Then

Trudy waits until Alice generates her next request to talk to Bob. That will be 008, “Alice”, “Bob”, KAlice{NA,008,“Alice”,“Bob”}. Trudy forwards that, along with the message KBob{NB,008,“Alice”,“Bob”} she recorded from Bob, to the KDC. The KDC will accept the messages now since both will contain 008. The only authentication of Bob in the Otway-Rees protocol is done by the KDC verifying that the nonce NC is the same in the message encrypted with KBob and the message encrypted with KAlice. Trudy can complete the exchange by forwarding to Alice the message the KDC sends her for Alice. Alice will wrongly assume the party to whom she is talking is Bob. If Alice and Bob are planning on using the shared key, say for encrypting the messages to one another, then Trudy won't have succeeded. But if all Alice and Bob want to do is authenticate, then Trudy will have successfully impersonated Bob to Alice. The Kerberos authentication service (see Chapter 11 Kerberos V4) is roughly based on the Needham-Schroeder protocol. It looks a lot simpler than these protocols because it assumes a universal idea of time, and includes expiration dates in messages. The basic Kerberos protocol is: 9.5 NONCE TYPES A nonce is a quantity which any given user of a protocol uses only once. Many protocols use nonces, and there are different types of nonces with different sorts of properties. It is possible to introduce security weaknesses by using a nonce with the wrong properties. Various forms of nonce are a timestamp, a large random number, or a sequence number. What’s different about these quantities? A large random number tends to make the best nonce, because it cannot be guessed or predicted (as Alice N1, Alice wants Bob KDC Bob 1 invents key KAB KAlice{N1,“Bob”,KAB,ticket to Bob} where ticket to Bob = KBob{KAB,“Alice”, expiration time} 2 ticket, KAB{T} where T is current time 3 KAB{T+1} 4 Protocol 9-21. Kerberos 9.5 NONCE TYPES 245 can sequence numbers and timestamps). This is somewhat unintuitive, since non-reuse is only probabilistic. But a random number of 128 bits or more has a negligible chance of being reused. A timestamp requires reasonably synchronized clocks. A sequence number requires nonvolatile state (so that a node can be sure it doesn’t use the same number twice even if it crashes and restarts between attempts). When are these properties important? Protocol 9-22 is a protocol in which the unpredictability of the challenge is important. Let’s say Bob is using a sequence number, and when Alice attempts to log in, Bob encrypts the next sequence number and transmits it to Alice, Alice decrypts the challenge and transmits it to Bob. Let’s say our eavesdropper Eve watches Alice’s authentication exchange, and sees Alice return an R of 7482. If Eve knows Bob is using sequence numbers for the challenge, she can then claim to be Alice, get an undecipherable pile of bits from Bob (the encrypted challenge), and return 7483. Bob will be suitably impressed and assume he’s talking to Alice. So it is obvious in this protocol that Bob’s challenge has to be unpredictable. How about if we do it the other way, i.e., make Alice do the encryption as in Protocol 9-23? Must the challenge then be unpredictable? Let’s say again that Bob is using sequence numbers. Eve watches Alice’s authentication exchange and sees that R is 7482. Then Eve lies in wait, impersonating Bob’s network address, hoping to entrap Alice into authenticating herself to Eve. When she does, Eve sends her the challenge 7483, and Alice will return the encrypted 7483. Now Eve can impersonate Alice to Bob, since Bob’s challenge will be 7483, and Eve will know how to encrypt that. This is a lot like a bucket brigade attack (see §6.4.1 The Bucket Brigade/Man-in-the-Middle Attack), to which any authentication-only protocol is vulnerable, but this is somewhat worse. In the Alice I’m Alice Bob KAlice-Bob{R} R Protocol 9-22. Protocol in which R must be unpredictable Alice I’m Alice Bob R KAlice-Bob{R} Protocol 9-23. Another protocol in which R must be unpredictable 246 SECURITY HANDSHAKE PITFALLS 9.6 bucket brigade attack, Eve has to impersonate Bob’s address to Alice and Alice’s address to Bob while both Alice and Bob are available on the network. In contrast this attack allows Eve to impersonate Bob’s address to Alice when Bob is down (and likewise Alice’s address when Alice is down). In many cases it is easier to impersonate someone's address if that someone is not available on the network. These protocols are also insecure if timestamps are used. Eve has to guess the timestamp Bob will use, and she might be off by a minute or two. If the timestamp has coarse granularity, say seconds, Eve has a good chance of being able to impersonate Alice. If the timestamp has, say, nanosecond granularity, then it really does become just like a random number and the protocol is secure. Here's a protocol in which it would be perfectly secure to use a predictable nonce for R. Even if Eve could predict what R would be, she can't predict either the value sent by Bob or the appropriate response from Alice. Getting truly random numbers can be expensive. A common method of obtaining pseudorandom numbers is by using a key known only to the node generating the numbers, and encrypting any sort of nonce, say a sequence number or timestamp. 9.6 PICKING RANDOM NUMBERS The use of pseudo-random processes to generate secret quantities can result in pseudo-security. —Richard Pitkin You can have perfect cryptographic algorithms, and perfectly designed protocols, but if you can't select good random numbers, your system can be remarkably insecure. Random numbers may be required in choosing cryptographic keys, challenges, IVs, or per-message secrets for ElGamal/DSS signatures (see §6.5.5 Per-Message Secret Number). Although I3'd love to rigorously define "random", I2 won't let me3 because a really rigorous definition is beyond the scope and spirit of the book. For instance, in [KNUT69], fifteen pages are Alice I'm Alice Bob KAlice-Bob{R} (KAlice-Bob+1){R} Protocol 9-24. Protocol in which R need not be unpredictable 9.6 PICKING RANDOM NUMBERS 247 devoted to coming up with a definition of a random sequence. For those readers who are not intimidated by notions such as (m,k)-distributed, serial correlation coefficients, and Riemann-integrability (don't try to find those terms in our glossary), the discussion of randomness in [KNUT69] is actually very interesting. The distinction is often made between random and pseudorandom. A pseudorandom number generator is a deterministic algorithm. The entire sequence it will generate is determined by its initial state. A random number generator is one that truly picks numbers unpredictably. It is difficult to have a source of true randomness. It can be done with special hardware that does something like measure the low order bits of a counter that counts radioactive particles per unit time. Although such hardware would not be difficult or expensive to provide [DAVI84b], most computers do not have it, so the typical technique is to use a pseudorandom number generator. Sometimes computers use the people they interact with as a source of randomness, for instance, by timing their keystrokes. Such a technique is useful to obtain a few bits of randomness, perhaps to seed a pseudorandom number generator. Applications that use random numbers have different requirements. For most applications, for instance one that generates test cases for debugging a computer program, all that might be required is that the numbers are spread around with no obvious pattern. For such applications it might be perfectly reasonable to use something like the digits of π. However, for cryptographic applications such as choosing a key, it is essential that the numbers be unguessable. Consider the following pseudorandom number generator. It starts with a truly random seed, say by timing a human's keystrokes. Then it computes a hash of the seed, then at each step it computes a hash of the output of the previous step. Assuming a good hash function, the output will pass any sort of statistical tests for randomness, but an intruder that captures one of the intermediate quantities will be able to compute the rest. It is possible to make a pseudorandom number generator that is as good as a source of true random numbers provided that it can be provided with an adequately unguessable seed. For example, a generator that outputs the hash of the seed, and thereafter outputs the hash of the previous output concatenated with the seed would be adequate provided, of course, that the seed was sufficiently unguessable. Implementations have made some amusing mistakes. Typical mistakes are: • seeding the generator with a seed that is from too small a space, say 16 bits. Suppose each time a cryptographic key needed to be chosen the application obtained 16 bits worth of true randomness from special purpose hardware and used those 16 bits to seed a pseudorandom number generator. The problem is, there would be only 65536 possible keys it would ever choose, and that is a very small space for an adversary (equipped with a computer) to search. Jeff Schiller found an application that used this technique with a seed that was only 8 bits! (The application has since been fixed.) 248 SECURITY HANDSHAKE PITFALLS 9.7 • using a hash of the current time when an application needs a random value. The problem is, some clocks do not have fine granularity, so an intruder that knew approximately when the program was run would not need to search a very large space to find the exact clock value for the seed. For instance, if a clock has 1/60 second granularity and the intruder knew the program chose the user's key somewhere within a particular hour, there would only be 60×60×60 = 216000 possible values. • divulging the seed value. An implementation, again discovered by Jeff Schiller (who warned the implementers so that the implementation has been fixed), used the time of day to choose a per-message encryption key. The time of day in this case may have had sufficient granularity, but the problem was that the application included the time of day in the unencrypted header of the message! In general the subroutines provided in programming languages for "random numbers" are not designed to be unguessable. Instead they are designed merely to pass statistical tests. A good strategy for finding a suitable seed for a pseudorandom number generator is to hash together all the sources of randomness you can find, e.g., timing of keystrokes, disk seek times, count of packet arrivals, etc. If any of the sources are unguessable the result will be. Even better, if there is any access to a good quality secret, hashing in the secret will make the seed unguessable. Note that if the secret is not of good quality, this strategy can backfire and give the attacker who learns one of your random numbers (say for an IV) enough information to do off-line password guessing. For more reading on the subject of randomness, see RFC 1750. The generation of random numbers is too important to be left to chance. Robert Coveyou, Oak Ridge National Laboratory 9.7 PERFORMANCE CONSIDERATIONS In addition to the security issues in the design of authentication protocols, there are also performance considerations. As CPUs get faster, performance becomes less important, but we haven't yet reached the point where it can be ignored. Metrics to be considered in evaluating the performance of an authentication protocol include: • number of cryptographic operations using a private key • number of cryptographic operations using a public key • number of bytes encrypted or decrypted using a secret key 9.8 AUTHENTICATION PROTOCOL CHECKLIST 249 • number of bytes to be cryptographically hashed • number of messages transmitted Note that public key operations are typically counted in blocks, while secret key and message digest operations are counted in bytes. The reason for this is that public key operations are so expensive that large amounts of data are not processed with public key operations. Public key operations are typically done on something that fits within a block, such as a secret key or a message digest of a message. For RSA, the de facto standard public key cryptographic algorithm, private key operations are much more expensive than public key operations. Other public key algorithms have different rules. Most secret key algorithms have similar performance for encrypting and decrypting. As a general rule, computing a message digest has somewhat better performance (per byte) than secret key encryption, but there is not a dramatic difference. Besides the processing power cost of cryptographic operations, there is also the real-time cost of performing an authentication. If an authentication protocol is running over a link to Mars, the number of round-trip message exchanges will certainly be the dominant factor in authentication time (as opposed to the processing time required to do the cryptographic operations). As processors get faster and the speed of light stays the same, the number of round-trip exchanges grows in relative importance as a factor in authentication time. Another consideration in real-time calculations is that if cryptographic operations can go on in parallel at the two ends of a connection, they will take less real time to complete. Some protocols are designed with this in mind (see §21.5.5 Lotus Notes Authentication, for example). Perhaps the most important optimization that can be added to a protocol is the ability to cache state from a previous authentication to make it easier for the same two parties to authenticate a second time. If a public key based authentication establishes a shared secret key as a side effect, that shared secret key can be used to do a more efficient authentication protocol on a second round if both ends remember it (see §14.9 Session Resumption). In a KDC-based scheme, it may be possible for one of the parties to remember a ticket from a previous authentication and thus avoid having to interact with the KDC on a second authentication (see §11.2 Tickets and Ticket-Granting Tickets). 9.8 AUTHENTICATION PROTOCOL CHECKLIST By way of review, the following is a checklist of things to consider when designing or evaluating an authentication protocol. Assume that Alice initiates conversations with Bob, and that Alice and Bob each have a database of information (some secret, some not) used in running the protocol. The list 250 SECURITY HANDSHAKE PITFALLS 9.8 is organized around what intruder Trudy might attempt to do and what the protocol being designed should protect against. • Eavesdrop. Even if Trudy can watch the messages between Alice and Bob pass over the network, she shouldn’t be able to ♦ learn the contents of the messages between Alice and Bob ♦ learn information that would enable her to impersonate either Alice or Bob in a subsequent exchange ♦ learn information that would enable her to impersonate Alice to another replica of Bob ♦ learn information that would permit her to do an off-line password-guessing attack against either Alice’s or Bob’s secret information • Initiate a conversation pretending to be Alice. Trudy can send a message to Bob and claim to be Alice and proceed at least partway through an authentication exchange. In doing so, she shouldn’t be able to ♦ convince Bob she is Alice ♦ learn information that would enable her to do an off-line password-guessing attack against either Alice’s or Bob’s secret information ♦ learn information that would enable her to impersonate Alice on a subsequent (or interleaved) attempt ♦ learn information that would enable her to impersonate Bob to Alice ♦ trick Bob into signing or decrypting something • Lie in wait at Bob’s network address and accept a connection from Alice. Trudy can get at least partway through an authentication exchange. In doing so, she shouldn’t be able to ♦ convince Alice that Trudy is Bob ♦ learn information that would enable her to do an off-line password-guessing attack against either Alice’s or Bob’s secret information ♦ learn information that would enable her to impersonate Bob on a subsequent attempt ♦ learn information that would enable her to impersonate Alice to Bob ♦ trick Alice into signing or decrypting something • Read Alice’s database. If Trudy can get all of Alice’s secrets, she can convince Bob she is Alice; she can also conduct an off-line password-guessing attack against Bob’s secret information (assuming Bob’s secret is derived from a password), since Alice must have enough information to know if someone is really Bob. She should not, however, be able to 9.8 AUTHENTICATION PROTOCOL CHECKLIST 251 ♦ impersonate Bob to Alice ♦ decrypt old recorded conversations between Alice and Bob • Read Bob’s database. If Trudy can get all of Bob’s secrets, she can convince Alice she is Bob; she can also conduct an off-line password-guessing attack against Alice’s secret information since Bob must have enough information to know if someone is really Alice. She should not, however, be able to ♦ impersonate Alice to Bob ♦ decrypt old recorded conversations between Alice and Bob • Sit on the net between Alice and Bob (e.g. as a router) and examine and/or modify messages in transit between them. Trudy can clearly prevent Alice and Bob from communicating, but she shouldn’t be able to ♦ learn information that would permit her to do an off-line password-guessing attack against either Alice's or Bob's secret information ♦ learn the contents of Alice's and Bob's messages ♦ hijack a conversation (continue a conversation started up by a legitimate party without the other side noticing the change) ♦ modify messages or rearrange/replay/reverse the direction of messages, causing Alice and/or Bob to misinterpret their messages to one another • Some combination of the above. There are clearly too many possibilities to enumerate, but some examples would be ♦ Even after reading both Alice's and Bob's databases, it should not be possible to decrypt old recorded conversations between them. ♦ Even after reading Bob's database and eavesdropping on an authentication exchange, it should not be possible to impersonate Alice to Bob. It isn't necessarily the right engineering trade-off to devise a protocol that passes all the criteria we mention above. For instance, in some cases eavesdropping or address impersonation might be a sufficiently unlikely threat that a much simpler protocol will suffice. Also, sometimes secrets are chosen from a large enough space that password guessing is not feasible. 252 SECURITY HANDSHAKE PITFALLS 9.9 9.9 HOMEWORK 1. Suppose Trudy hijacks a conversation between Alice and Bob. This means that after the initial handshake, Trudy sends messages with source address equal to Alice's source address. Suppose the network allows Trudy to insert a fake source address (Alice's source address), but does not deliver packets destined for Alice's address to Trudy. What are the problems involved in having Trudy transmit a file to Bob as if she were Alice? Consider potential problems with flow control and file transfer protocols when Trudy cannot see return traffic from Bob. 2. In §9.2 Mutual Authentication, we discuss the reflection attack and note that Protocol 9-8 is susceptible, but Protocol 9-7 is not. How about Protocol 9-11? 3. In §9.3.1 Shared Secret we discuss various possibilities for forming a session key. Remember that R is the challenge sent by Bob to Alice, and A is Alice's secret, which Bob also knows. Which of the following are secure for a session key? $A \oplus R$ $\{R+A\}A$ $\{A\}A$ $\{R\}R+A$ 4. Design a variant of Otway-Rees (page 243) that only has one nonce generated by Alice and one nonce generated by Bob. Explain why it is still as secure. 5. Suppose we are using a three-message mutual authentication protocol, and Alice initiates contact with Bob. Suppose we wish Bob to be a stateless server, and therefore it is inconvenient to require him to remember the challenge he sent to Alice. Let's modify the exchange so that Alice sends the challenge back to Bob, along with the encrypted challenge. So the protocol is: Is this protocol secure? Alice I'm Alice Bob R R, KAlice-Bob{R} 9.9 HOMEWORK 253 6. Let's modify the protocol from the previous problem so that Bob sends both a challenge, and a challenge encrypted with a key that only he knows, to Alice: Is this protocol secure? 7. In the discussion of Protocol 9-3 on page 225, Bob remembers all the timestamps he's seen within the last 10 minutes. Why is it sufficient for him to remember only 10 minutes worth of timestamps? 8. Design a two-message authentication protocol, assuming that Alice and Bob know each other's public keys, which accomplishes both mutual authentication and establishment of a session key. 9. The Expanded Needham-Schroeder Protocol (page 242) can be shortened to a 6-message protocol without loss of security by merely removing the 7th message. Why is this true? (Hint: the purpose of the 7th message is to prove to Bob that he is talking to Alice, but he already knows that. Why?) 10. §9.4 Mediated Authentication (with KDC) describes several protocols. For each of those protocols, describe which nonces have to be unpredictable (i.e., sequence numbers would not be good). 11. As we pointed out in §7.1 Password-Based Authentication, cellular phones are vulnerable to a fraud known as "cloning". The protocol cellular phones use is that a phone transmits its telephone number followed by a cleartext password. The phone company checks its database of phone number/password to make sure the phone is legitimate before allowing the call to go through. The phone number is the one billed. Suggest a design based on public key, and one based on secret key, technology. Can you guard against the phone company database being stolen? 12. There is a product which consists of a fancy telephone that, when talking to a compatible fancy telephone, does a Diffie-Hellman key exchange in order to establish a secret key, and the remainder of the conversation is encrypted. Suppose you are a wiretapper. How can you listen to a conversation between two such telephones? Alice I'm Alice Bob R, KBob{R} KBob{R}, KAlice-Bob{R} 254 SECURITY HANDSHAKE PITFALLS 9.9 13. In §9.1.1 Shared Secret, we discussed using MD5(KAlice-Bob|R) as the method of encrypting R with KAlice-Bob. (When we say KAlice-Bob|R we mean KAlice-Bob concatenated with R.) Suppose instead we used MD5(KAlice-Bob∨R). Would that be secure? How about MD5(KAlice-Bob⊕R)? 255 10 STRONG PASSWORD PROTOCOLS 10.1 INTRODUCTION Suppose a user, Alice, wants to use any workstation to log into a server, Bob. Assume she has nothing but a password with which to authenticate herself. Assume the workstation has no user-specific configuration, such as the user's trusted CAs, or the user's private key. Also assume that the software on the workstation is trustworthy. There are various ways Alice might use a password to authenticate herself to server Bob: • Transmit it over the wire, in the clear. This leaves Alice's password vulnerable to discovery by an eavesdropper, or someone impersonating Bob. • Do an anonymous Diffie-Hellman exchange to establish a secret key and an encrypted tunnel, and send the password over that encrypted tunnel. This protects Alice's password from a passive attacker, but not from someone impersonating Bob. • Create an SSL connection, using the trust anchors configured into the client machine and a server certificate issued by one of those trust anchors, so that Alice's machine can authenticate Bob, and then send the password SSL-encrypted to Bob. This relies on the trust anchors at Alice's machine being configured properly, and none of the trust anchor organizations having inadvertently (or maliciously) certified a bogus public key as being Bob's. • Compute a hash of Alice's password, and use that as a secret key in a challenge/response authentication handshake—Bob sends challenge R, Alice responds with f(password, R). This leaves Alice's password vulnerable to a dictionary attack, either by an eavesdropper or someone impersonating Bob. • Use a one time password scheme like Lamport's hash or S/KEY (see §10.2 Lamport's Hash). • Use a strong password protocol (see §10.3 Strong Password Protocols). Strong password protocols are secure even if the shared secret, typically a password, could be broken by an off- 256 STRONG PASSWORD PROTOCOLS 10.2 line dictionary attack. The exchange is carefully designed to prevent the opportunity for either a passive attacker or someone impersonating either side to obtain data with which to do a dictionary attack. 10.2 LAMPORT'S HASH It's a poor sort of memory that only works backwards. —The White Queen (in Through the Looking Glass) Leslie Lamport invented an interesting one-time password scheme [LAMP81]. This scheme allows Bob to authenticate Alice in a way that neither eavesdropping on an authentication exchange nor reading Bob's database enables someone to impersonate Alice, and it does it without using public key cryptography. Alice (a human) remembers a password. Bob (the server that will authenticate Alice) has a database where it stores, for each user: • username • n, an integer which decrements each time Bob authenticates the user • hashn (password), i.e., hash(hash(...(hash(password))...)) First, how is the password database entry associated with Alice configured? Alice chooses a password, and a reasonably large number n (like 1000) is chosen. The user registration software computes x1=hash(password). Then it computes x2=hash(x1). It continues this process n times, resulting in xn=hashn(password), which it sends to Bob, along with n. When Alice wishes to prove her identity to Bob, she types her name and password to her workstation. The workstation then sends Alice's name to Bob, which sends back n. Then the workstation computes hashn−1(password) and sends the result to Bob. Bob takes the received quantity, hashes it once, and compares it with its database. If it matches, Bob considers the response valid, replaces the stored quantity with the received quantity, and replaces n by n−1. If n ever gets to 1, then Alice needs to set her password again with Bob. There is no completely secure way of doing this over an insecure network, since this scheme does not allow encryption or integrity protection of messages between Alice and Bob. But in practice, in many situations, it suffices for Alice to choose a new password, compute hashn(new password), and transmit hashn (new password) and n to the server unencrypted across the network. An enhancement is to add salt, a number chosen at password installation time to be unique for user Alice. The salt is stored at Bob and concatenated to the password before hashing. So rather than computing hashn(password), the enhanced Lamport hash computes hashn (password|salt). To 10.2 LAMPORT'S HASH 257 set the password, the workstation chooses a value for salt, and computes x1=hash(password|salt), then x2=hash(x1), then continues this process n times, resulting in xn=hashn(password|salt), which it sends to Bob, along with n and salt. What do we gain by adding salt? It allows Alice to securely use the same password on multiple servers as long as a different salt value is used when installing the password on each of the other servers. When she logs into Bob, she'll wind up decrementing the n stored at Bob, but this will not affect the n stored at other servers. In other words, when she logs into Bob, if Bob sends her workstation ⟨n=87, salt=69⟩, her workstation will compute hash86(password|69). If when Bob hashes the received quantity the result matches hash87(password|69) in his database, then Bob will decrement n and replace the stored hash with the received hash. When she logs into Fred, Fred might send her workstation ⟨n=127, salt=105⟩, in which case her workstation will compute hash126(password|105) to send to Fred. A way of ensuring that the salt is different on different servers is to also hash in the server name, as in xn=hashn(password|salt|servername). Another advantage of salt is that Alice does not need to change her password when n decrements to 1 at Bob. Instead the same password can be reinstalled with a different salt value. There's an additional value to salt, which is the same as the original UNIX reason for adding salt to the password database (see §5.2.4.1 UNIX Password Hash). Adding salt prevents an intruder from precomputing hashk for all passwords in a dictionary and all values of k from 1 through 1000, stealing the database at Bob, and then comparing the precomputed hashes with the stolen password hashes of all the users. Lamport's hash has interesting properties. It is similar to public key schemes in that the database at Bob is not security sensitive (for reading), other than dictionary attacks to recover the user's password. It has several disadvantages relative to public key schemes. One problem is that you can only log in a finite number of times before having to reinstall password information at the server. Another problem is there is no mutual authentication, i.e., Alice does not know she is definitely talking to Bob. This makes it difficult to establish a session key or prevent a man-in-the-middle attack. One might try to have Alice authenticate herself and then do a Diffie-Hellman exchange with Bob to establish a session key. But Trudy could hijack the conversation after the inital authen- Alice Alice, pwd Alice's Workstation Alice Bob knows ⟨n, hashn(password)⟩ n x = hashn−1(pwd) compares hash(x) to hashn(password); if equal, replaces ⟨n, hashn(password)⟩ with ⟨n−1, x⟩ Protocol 10-1. Lamport's Hash 258 STRONG PASSWORD PROTOCOLS 10.2 tication and before the Diffie-Hellman exchange. Another idea might be for Alice and Bob to do the Diffie-Hellman exchange first and then do the authentication handshake protected with the DiffieHellman key. But Trudy could act as a man-in-the-middle, establishing a separate Diffie-Hellman key with each of Alice and Bob, and simply relay the authentication handshake. Once Alice sends the authentication information, Trudy can break her connection with Alice and continue conversing with Bob, impersonating Alice. There's another security weakness, which we'll call the small n attack. Suppose an intruder, Trudy, were to impersonate Bob's network address and wait for Alice to attempt to log in. When Alice attempts to log into Bob, Trudy sends back a small value for n, say 50 (and Alice's salt value at Bob, which she can know from having eavesdropped on a previous authentication by Alice to Bob). When Alice responds with hash50(password), Trudy will have enough information to impersonate Alice for some time, assuming that the actual n at Bob is greater than 50. What can be done to protect against this? Alice's workstation could display n to the human Alice. If Alice remembers approximately what n should be, then Alice can do a rough sanity check on n. Lamport's hash can also be used in environments where the workstation doesn't calculate the hash, for example when: • Alice is logging in from a "dumb terminal" • Alice is logging in from a workstation that does not have Lamport hash code, or • Alice is logging in from a workstation that she doesn't trust enough to tell her password We'll call this the human and paper environment, and call the other environment the workstation environment. The way Lamport's hash works in the human and paper environment is that when the information ⟨n,hashn (password)⟩ is installed at the server, all the values of hashi (password) for i< p. How could the other side prevent this? 7. Show protocols for doing augmented forms of EKE and SPEKE. 8. Show how Alice computes hash(2ab mod p, 2bW mod p) in Protocol 10-3. 9. Show how in Protocol 10-3 Alice can be assured that it is Bob, i.e., that the other side has the information stored at Bob. Explain why someone who has stolen Bob's database cannot impersonate Alice to Bob. 10. (*) Suppose in message 3 in Protocol 10-4, Alice signs only c. What vulnerability would this have? (Hint: someone who has captured Bob's database can do a man-in-the-middle attack.) 11. Explain how each of Alice and Bob compute K in the SRP protocol (Protocol 10-4). 12. (*) In Protocol 10-4, Bob's Diffie-Hellman value gb mod p is not encrypted with W. Argue why this is still secure. Note that it would not be secure for Bob to send gb mod p without encrypting it with W in Protocol 10-2. Why is that? How could you modify Protocol 10-2 so that it would be secure for Bob not to encrypt gb mod p with W? 13. Show credentials download protocols built upon SPEKE, PDM, and SRP. 14. Why is the EKE-based Protocol 10-7 insecure? (Hint: someone impersonating Bob can do a dictionary attack, but show how.) How can you make it secure while still having Bob transmit gb mod p unencrypted? 15. (*) Consider Protocol 10-8. How would Alice compute K? How would Bob compute K. Why is it insecure? (Hint: someone impersonating Bob can do a dictionary attack, but show how.) 10.5 HOMEWORK 267 Alice Bob stores "Alice", W choose random a "Alice", W{ga mod p} choose random b and challenge c g 1 b mod p, c1 compute W from password K = gab mod p choose challenge c2 K{c1}, c2 K{c2} Protocol 10-7. For Homework Problem 14 choose a Alice Bob stores "Alice", gW mod p "Alice", ga mod p choose b and challenge c g 1 b mod p, c1 compute W from password choose challenge c2 K = hash(gab mod p, gWb mod p) K{c1}, c2 K{c2} Protocol 10-8. For Homework Problem 15 PART 3 STANDARDS 269 11 KERBEROS V4 11.1 INTRODUCTION Kerberos is a secret key based service for providing authentication in a network. When a user Alice first logs into a workstation, she'll type her account name and password. We'll call the period from when she logs in to when she logs out her login session. During her login session Alice will probably need to access remote resources (such as hosts on which Alice has accounts, or file servers on which Alice has files). These remote resources will need to authenticate her, but Alice's workstation performs the authentication protocol on Alice's behalf, and Alice need not be aware that it is happening. The network itself is assumed to be insecure. Bad guys might eavesdrop or modify messages. Kerberos was originally designed at MIT. It is based on work by Needham and Schroeder (see § 9.4 Mediated Authentication (with KDC)). The first three versions of Kerberos are no longer in use and hence are not of much interest. Version 4 and Version 5, although conceptually similar, are substantially different from one another and are competing for dominance in the marketplace. Version 4 has a greater installed base, is simpler, and has better performance, but works only with TCP/IP networks, while Version 5 has greater functionality. We'll describe Version 4 in detail in this chapter, and the differences between Version 4 and Version 5 in the next. For any inquiring minds out there, the name Kerberos is the name of the three-headed dog that guards the entrance to Hades. (Wouldn't it be more useful to guard the exit?) An implementation of Kerberos consists of a Key Distribution Center (KDC) that runs on a physically secure node somewhere on the network, and a library of subroutines that are used by distributed applications which want to authenticate their users. While there are many possible modes of operation, Kerberos was designed for an environment where a user logs into a workstation by providing a user name and a password. These are used by the workstation to obtain information from the KDC that can be used by any process to access remote resources on behalf of the user. The basic ideas behind Kerberos should be familiar by now, since we've already discussed the concept of KDCs in §7.7 Trusted Intermediaries and § 7.4 Mediated Authentication (with KDC). Some applications that have been modified to call subroutines in the Kerberos library as part of their startup include 270 KERBEROS V4 11.2 • telnet (RFC 854)—a protocol for acting as a terminal on a remote system • BSD rtools—a group of utilities included with BSD UNIX that support remote login (rlogin), remote file copying (rcp), and remote command execution (rsh) • NFS (Network File System, RFC 1094)—a utility that permits files on a remote node on the network to be accessed as though they were local files 11.2 TICKETS AND TICKET-GRANTING TICKETS The KDC shares a secret key, known as a master key, with each principal (each user and each resource that will be using Kerberos). When Alice informs the KDC that she wants to talk to Bob, the KDC invents a session key KAB for Alice and Bob to share, encrypts KAB with Alice's master key for Alice, encrypts KAB with Bob's master key for Bob, and returns all this information to Alice. The message consisting of the session key KAB (and other information such as Alice's name) encrypted with Bob's master key is known as a ticket to Bob. Alice can't read what's inside the ticket, because it's encrypted with Bob's master key. Bob can decrypt the ticket and discover KAB and Alice's name. Bob knows, based on the ticket, that anyone else who knows KAB is acting on Alice's behalf. So Alice and Bob can authenticate each other, and optionally encrypt or integrityprotect their entire conversation, based on the shared key KAB. The session key KAB together with the ticket to Bob are known as Alice's credentials to Bob. Alice logs into a workstation by supplying her name and password. Alice's master key is derived from her password. The workstation, which performs the authentication protocol on behalf of Alice, could remember her password during the whole login session and use that when proof of her identity was required. But that is not good security practice for various reasons. One worry is that Alice might, during the login session, run untrustworthy software that might steal her password. To minimize harm, the first thing that happens is that the workstation asks the KDC for a session key SA for Alice to use for just this one session. The session key SA is then used on Alice's behalf when asking for tickets to resources on the network. The key SA is only valid for some small amount of time, generally a few hours. If anyone steals SA, then they can impersonate Alice, but only until SA expires. During the initial login, the workstation, on Alice's behalf, asks the KDC for a session key for Alice. The KDC generates a session key SA, and transmits SA (encrypted with Alice's master key) to the workstation. The KDC also sends a ticket-granting ticket (TGT), which is SA (and other information such as Alice's name and the TGT's expiration time) encrypted with the KDC's master key. 11.3 CONFIGURATION 271 The workstation uses Alice's master key (derived from her password) to decrypt the encrypted SA. Then the workstation forgets Alice's password and only remembers SA and the TGT. When Alice later needs to access a remote resource, her workstation transmits the TGT to the KDC, along with the name of the resource to which Alice needs a ticket (say Bob). The KDC decrypts the TGT to discover SA, and uses SA to encrypt the

Alice-Bob session key for Alice. Essentially, the TGT informs the KDC to use SA instead of Alice's master key. Earlier versions of Kerberos did not have the notion of a TGT. The user's master key was always used when communicating with the KDC. Later, for enhanced security, the notion of obtaining a limited-lifetime session key at login time was added. The idea was that only TGTs were gotten from the KDC, and that other tickets were obtained from what was theoretically a different entity, which Kerberos calls a Ticket-Granting Server or TGS. However, the Ticket-Granting Server has to have the same database as the KDC (it needs to know Bob's secret key in order to give Alice a ticket for Bob). So in Kerberos, the Ticket-Granting Server and the KDC are really the same thing. It would be easier to understand Kerberos if the documentation didn't refer to two different entities (KDC and TGS) and two sets of protocol messages that basically do the same thing. The reason it is described that way in Kerberos documentation is solely due to how the protocol evolved. The Kerberos documentation refers to an Authentication Server (AS) and a Ticket-Granting Server (TGS) which are collocated at the KDC, meaning they are really the same thing. We will use the terms KDC, AS, and TGS interchangeably, since we do not find the distinction to be useful for understanding Kerberos, and in reality there is no distinction. 11.3 CONFIGURATION As we said before, each principal has its own secret key, known as the master key for that principal. The Kerberos server is called a KDC (Key Distribution Center). Sometimes the Kerberos documentation calls the KDC an Authentication Server, or a KDC/AS. The KDC has a database of names of principals and their corresponding master keys. To keep the KDC database reasonably secure, the stored master keys are stored encrypted by a key that is the personal secret of the KDC, known as the KDC master key. The master key for a human user is derived from the user's password. Resources other than human beings are configured with their own master keys. Kerberos is based on secret key technology, and all current implementations use DES. Kerberos V5 has fields in packets for identifying the cryptographic algorithm, so theoretically an algorithm other than DES can be used in V5, but in reality we can assume the algorithm is DES because there are no implementations of Kerberos with the ability to use a different algorithm. 272 KERBEROS V4 11.4 To summarize, human users need to remember a password. Other network devices need to remember a secret key. The KDC needs to have a database of names of all the principals for which it is responsible, together with a secret key for each of them. It also needs a secret key KKDC for itself (in order to encrypt the database of user keys and in order to generate ticket-granting tickets). 11.4 LOGGING INTO THE NETWORK By way of introduction, let's look at the steps that take place as Alice uses the network. She'll expect the login procedure to be pretty much the same as for logging into an operating system. First, the workstation prompts Alice for a name and a password. 11.4.1 Obtaining a Session Key and TGT Alice types her account name and password at the workstation. The workstation sends a message to the KDC, in the clear (unencrypted), which gives Alice's account name. On receipt of the request, the KDC returns credentials to the KDC, encrypted with Alice's master key. The credentials consist of • a session key SA (a secret key to be used during the login session). • a ticket-granting ticket (TGT). The TGT contains the session key, the user's name, and an expiration time, encrypted with KKDC. Because it is encrypted with the KDC's master key, the TGT is an unintelligible bunch of bits to anyone other than the KDC. These credentials are sent back to the workstation encrypted with Alice's master key, KA. Note that the information in the TGT is therefore doubly encrypted when transmitted by the KDC—first with KKDC and then with KA. Kerberos V4 is sometimes criticized for this minor performance suboptimality, since encrypting the already-encrypted TGT offers no security benefit. The workstation converts the password Alice types into a DES key. When the workstation receives the credentials, it attempts to decrypt them using this DES key. If this decryption succeeds (which it will if Alice typed her password correctly), then the workstation discards Alice's master key (the key derived from her password), retaining instead the TGT and the session key. Kerberos documentation refers to the request sent by the workstation to the KDC as a KRB_AS_REQ, for Kerberos Authentication Server Request. We'll call it AS_REQ. The message in which the KDC returns the session key and TGT is known in the Kerberos documentation as a KRB_AS_REP, for Kerberos Authentication Service Reply. We'll call it AS_REP. 11.4.2 LOGGING INTO THE NETWORK 273 Actually, Kerberos V4 does not prompt the user for her password until after the workstation has received the credentials from the KDC. This is because Kerberos V4 was very serious in following the generally good security rule of having the workstation know the user's password for the minimum time possible. Waiting the few seconds to get the credentials before asking the user for the password really doesn't enhance security significantly, and in fact V5 has the user type the password before the workstation sends the AS_REQ. The reason V5 changed the order was that V5 requires the workstation to prove it knows the user's password before it sends the credentials, which makes it less easy to obtain a quantity with which to do off-line password guessing. An eavesdropper will still be able to do off-line password guessing with V5, but in V4 all you have to do is send the name Alice to the KDC and it will return a quantity with which you can do password guessing. This is likely to be easier than eavesdropping. What is the purpose of the TGT? When Alice needs to access a remote resource, her workstation sends the TGT to the KDC along with a request for a ticket to the resource's node. The TGT contains the information the KDC needs about Alice's login session (session key, Alice's name, expiration time,...). This allows the KDC to operate without having any volatile data; it has a largely static database, and for each request it sends a response and then forgets that it happened. This offers a number of operational advantages, like making it easy to replicate the KDC and not having to maintain state across crashes. An interesting variant might be to have the workstation generate the TGT (see Homework Problem 1). 11.4.2 Alice Asks to Talk to a Remote Node Suppose that after logging in as described above, Alice types a command that requires access to a remote node (like rlogin Bob, which logs Alice into Bob). The workstation sends to the KDC the TGT, the name Bob, and an authenticator which proves that the workstation knows the session key. The authenticator consists of the time of day encrypted with the session key (in this case SA). This request is known in the Kerberos documentation as a KRB_TGS_REQ, and the reply is Alice Alice,password Workstation [AS_REQ] Alice needs a TGT KDC invents key SA finds Alice's master key KA TGT = KKDC{"Alice",SA} [AS_REP] KA{SA,TGT} Figure 11-1. Obtaining a TGT 274 KERBEROS V4 11.4.2 known as KRB_TGS_REP; we'll call them TGS_REQ and TGS_REP. The TGS_REP contains a ticket to Bob and KAB (the session key to be shared by Alice and Bob), encrypted with SA (the session key to the KDC). Because of the use of authenticators it is necessary for resources on the network to keep reasonably synchronized time. The times can be off by some amount. The allowable time skew is independently set at each server, and therefore some servers may be configured to be fussier than others about times being close. The allowed time skew is usually set to be on the order of five minutes on the assumption that it is possible to get computers' clocks to be accurate within five minutes without undue administrative burden. In practice, that assumption has turned out to be more problematic than expected. Distributed time services, once deployed, make much tighter synchronization straightforward. It turns out that there is no security or functionality gained by having Kerberos require an authenticator when Alice's workstation requests a ticket to Bob. If someone who didn't know the session key transmitted the TGT and the name Bob to the KDC, the KDC would return information encrypted with SA, which would be of no use to someone who didn't know Alice's session key. The reason the designers of Kerberos did it this way is to make the protocol for talking to the TicketGranting Service of the KDC be the same as for talking to other resources. When talking to most resources other than the KDC, the authenticator does provide security, because it prevents the replay of old requests and authenticates the sender (which is important if the reply is unencrypted). The KDC decrypts the TGT (with KKDC) and discovers the session key SA. It also checks the expiration time in the TGT. If the TGT is valid, the KDC constructs a new key KAB, for use in talking between Alice and Bob, and constructs a ticket, which consists of the newly generated key KAB, the name Alice, and an expiration time, all encrypted with Bob's master key, KB. The KDC sends the ticket, along with the name Bob and KAB, to the workstation. Again this information must be encrypted, and it is encrypted with SA. On receipt, the workstation decrypts the information using SA. Now the workstation sends a request to Bob. In the Kerberos documentation this request is called a KRB_AP_REQ, for application request, which we'll call AP_REQ. It consists of the ticket and an authenticator (in this case the time encrypted with the session key KAB). The reply from Bob is known in Kerberos as KRB_AP_REP, and we'll call it AP_REP. Bob decrypts the ticket and discovers the key KAB and the name Alice. Bob now assumes that anyone with knowledge of KAB is acting on Alice's behalf. Then Bob decrypts the authenticator to know that the party to which he is speaking does indeed know the session key KAB. He checks that the time in the decrypted authenticator is close to current (within five minutes) to ensure that this is not a replay of some earlier request. To make sure it is not a replay of a request recent enough to look current given the time skew, Bob should keep all timestamps he has received recently, say in the last five minutes (a parameter set appropriately for the maximum allowable time skew) and check that each received timestamp from a given source is different from any of the stored values. Any authenticator older than five 11.4.2 LOGGING INTO THE NETWORK 276 minutes (or whatever the value of the maximum allowable time skew) would be rejected anyway, so Bob need not remember values older than 5 minutes. Kerberos V4 doesn't bother saving timestamps. Saving timestamps doesn't help anyway if Bob is a replicated service in which all the instances of Bob use the same master key. The threat of an eavesdropper replaying the authenticator Alice sent to one instance of Bob to a different instance of Bob could have been avoided if Kerberos had done something like put the network layer address of the instance of Bob in the authenticator. To provide mutual authentication, Bob adds one to the time he decrypted from the authenticator, reencrypts that with KAB and sends it back. Alice's workstation is now reassured that it is talking to Bob, since the party at the other side was able to decrypt the ticket, which meant he knew KAB, which was encrypted with KB. Thereafter, depending on the application, messages between Alice and Bob may be unprotected, integrity-protected, or encrypted and integrity-protected. Some applications always use the same Kerberos protection (authentication only, data integrity protection, or data encryption plus integrity protection). Others make it optional (a switch setting when calling the application). The decision is a security vs. performance trade-off when using Kerberos. Alice rlogin Bob Workstation [TGS_REQ] Alice wants to talk to Bob TGT = KKDC{“Alice”,SA} authenticator = SA{timestamp} KDC invents key KAB decrypts TGT to get SA decrypts authenticator verifies timestamp finds Bob's master key KB ticket to Bob = KB{“Alice”,KAB} [TGS_REP] SA{“Bob”, KAB, ticket to Bob} Figure 11-2. Getting a ticket to Bob for Alice Alice's Workstation [AP_REQ] ticket to Bob = KB{“Alice”,KAB} authenticator = KAB{timestamp} Bob decrypts ticket to get KAB decrypts authenticator verifies timestamp [AP_REP] KAB{timestamp+1} Figure 11-3. Logging into Bob from Alice's workstation 276 KERBEROS V4 11.5 11.5 REPLICATED KDCS A serious problem with having all authentication rely on a single KDC is that it is a single point of failure. If the KDC is unavailable, either because it is down or because the path through the network to the KDC is broken, it is impossible to access remote resources, making the network unusable to everyone. In addition, a single KDC might be a performance bottleneck, since all logins and all attempts to start conversations to anything must involve communication with the KDC. For these reasons it is desirable to have multiple KDCs, where each KDC is interchangeable with every other KDC. They share the same master KDC key and have identical databases of principal names and master keys. Keeping the databases at all the KDCs the same is done by having one site hold the master copy to which any updates must be made. An update consists of adding an entry for ⟨principal name, key⟩, modifying an entry (for instance to change a key), or deleting an entry. Other sites download the database periodically, either on a timer or as a result of a human issuing a command. Having a single master copy avoids problems such as combining updates made at different replicas and resolving conflicting updates. Of course, by having a single master copy to which updates must be made, Kerberos still has a single point of failure. Fortunately, Kerberos is designed so that most operations—and all critical operations—are read-only operations of the KDC database. It's true that if the master is down, administrators can't add and delete users (but when the KDC is down the administrators are probably too busy to bother with adding and deleting users since they're trying to fix the KDC), and users can't change their passwords (which may be inconvenient but is not life-threatening). These operations can wait for a few hours until the master copy is repaired. In contrast, the read-only KDC operations are required for any use of the network. If there were only a single KDC, and it was unavailable, none of the network users would be able to get any (computer-related) work done. Another reason for replication is to avoid a performance bottleneck at the KDC. Updates are sufficiently rare that the master copy will be easily capable of keeping up with all updates. Most of the operations will be read-only, and these operations will be spread among the read-only replicas. Another reason that replication may improve performance is so that a KDC replica is usually nearby. When downloading the KDC database from the master replica to a read-only slave, it is important to protect the data from disclosure and modification. Disclosure would permit an attacker to learn the master keys of all principals; modification would permit an attacker to create new accounts or change the properties of existing ones. This protection could have been provided by encrypting the KDC database as a unit when it was being transferred. The Kerberos designers chose not to do that. Because the principals' master keys are stored in the database encrypted under the KDC master key, there is no serious disclosure threat. An attacker could learn the names of all principals and 11.6 REALMS 277 their properties, but not their master keys. The threat remains that an attacker could rearrange data in transit so that, for example, some privileged user was given the attacker's master key (by simply copying the encrypted key field). This second threat is avoided by transferring the KDC database as a file in the clear but then sending a cryptographic hash of the file in a Kerberos protected exchange. By using the integrity protection of Kerberos, which includes a timestamp, the protocol prevents the attacker from substituting an old version of the KDC database, for instance one from before the attacker was fired. 11.6 REALMS Having a single KDC for an entire network is a problem for many reasons. With replicated KDCs, you can alleviate bottleneck problems and eliminate the single-point-of-failure problem. However, there is a serious problem which creation of redundant KDCs does not solve. Imagine a big network consisting of several organizations such as companies in competition with each other, various universities, banks, and government agencies. It would be hard to find an organization that everyone would trust to manage the KDC. Whoever manages the KDC can access every user's master key, and therefore access everything that every user can access. Furthermore, that highly trusted entity would also be a busy one, having to process all instances of users and services joining and leaving the network. Even if there were an organization everyone was willing to trust, this is not enough. Everyone must also trust the physical controls around every replica of the KDC, and there would have to be widely dispersed replicas to ensure availability and convenience.

Compromise of any replica, no matter how obscurely placed it was, would yield everyone's keys. For this reason the principals in the network are divided into realms. Each realm has its own KDC database. There can be multiple KDCs in a realm, but they would be equivalent, have the same KDC master key, and have the same database of principals' master keys. Two KDCs in different realms, however, would have different KDC master keys and totally different principals' master key databases, since they would be responsible for a different set of principals. In V4, a name of a principal has three components, each of which is a null-terminated casesensitive text string of up to 40 characters: NAME, INSTANCE, and REALM. The Kerberos protocol does not particularly care how the NAME and INSTANCE fields are used. They are merely text strings. However, in practice, services have a name, and the INSTANCE field is used to indicate the particular machine on which the service is running. For example the file service might be called fileserv and a particular instance might be jailbreak, for the name of one of the fileserver machines. 278 KERBEROS V4 11.7 The INSTANCE field is not as useful for human principals, and usually the null string would be used for the instance part of the name. But as long as there is that component to a name, the convention for humans is that it is used to denote something about the role of the human for that particular login session. For example, someone can have the name Alice and the instance systemmanager, or the name Alice and the instance gameplayer. Presumably Alice.systemmanager would have access to many more network resources than Alice.gameplayer. In effect, a name is really a concatentation of the two strings NAME and INSTANCE. Kerberos could certainly have done without the concept of instance, and by convention principals could just use different names in their various roles. As a matter of fact, if "." were a legal character in the text string for a name, the names could be identical to what they are currently, without having to separate them officially into two fields (NAME and INSTANCE). 11.7 INTERREALM AUTHENTICATION Suppose the world is partitioned into n different Kerberos realms. It might be the case that principals in one realm need to authenticate principals in another realm. This is supported by Kerberos. The way it works is that the KDC in realm B can be registered as a principal in realm A. This allows users in realm A to access realm B's KDC as if it were any other resource in realm A. Suppose Alice, in realm Wonderland, wishes to communicate securely to Dorothy in realm Oz. Alice (or rather her workstation, on her behalf) notices that Dorothy is in a different realm. She asks her KDC for a ticket to the KDC in realm Oz (see Figure 11-4). If the managers of Wonderland and Oz have decided to allow this, the KDC in Oz will be registered as a principal in Wonderland, and the Wonderland KDC will have assigned a master key to Oz's KDC. Wonderland's KDC will give Alice a ticket (encrypted with that master key) to the KDC in Oz. Then Alice sends a TGS_REQ to Oz's KDC. The message lists Wonderland as the source realm, which tells Oz's KDC what key to use to decrypt the ticket. The Oz KDC then issues a ticket for Alice to talk to Dorothy. After Alice sends this ticket to Dorothy, Alice and Dorothy will know they are talking to each other, and they will have a key KA-D they can use to protect data they send back and forth, exactly as if they were in the same realm. It doesn't work in Kerberos V4 to start in realm A, get a ticket to realm B, and from there get a ticket to realm C. In order for a principal in realm A to talk to a principal in realm C, C's KDC has to be registered as a principal in realm A. What prevents going from realm A to C through B? Suppose realms A and B share a key, and realms B and C share a key, but A and C do not share a key. Suppose Alice@A would like to talk to Carol@C. Alice can indeed get a ticket to B, since B is registered as a principal in realm A. Then, with a ticket to B, Alice can request, from B, a ticket to C. B 11.8 KEY VERSION NUMBERS 279 will comply, since B thinks of C as just another principal in realm B. But then when A attempts to ask C for a ticket to Carol, the TGS_REQ she sends C will have the REALM field as B (so that C will know what key to use to decrypt the ticket), and in the ticket, Alice's realm will be A. C will refuse to issue a ticket for Carol since the two realms don't match. (See §11.12.5 TGS_REQ for a description of the fields in the TGS_REQ.) Kerberos V4 deliberately prevents access through a chain of KDCs. If it didn't, then a rogue KDC could impersonate not only its own users, but those of any other realm, by claiming to be the penultimate KDC in a chain. 11.8 KEY VERSION NUMBERS In most operating systems, Alice can change her password without inconveniencing anyone else. The Kerberos designers wanted the same to be true when using Kerberos. However, with Kerberos, a change in a principal's master DES key or a change in the KDC's master DES key would affect tickets held by other users. For instance, if Alice has a ticket to Bob, and Bob changes his master DES key, Alice's ticket will still be encrypted with Bob's old master DES key. Similarly, if Alice has a TGT with a session key for her login session, and the KDC's master key is changed, then Alice's TGT will still be encrypted with the old KDC master key. The Kerberos designers could simply have ignored the problem, and then communication would fail. Presumably principals would at that point ask for a new ticket and things would pro- Alice TGS_REQ("Alice@Wonderland","Oz@Wonderland") Wonderland KDC Oz KDC Dorothy credentials to Oz TGS_REQ("Alice@Wonderland","Dorothy@Oz") credentials to Dorothy AP_REQ Figure 11-4. Interrealm authentication 280 KERBEROS V4 11.9 ceed. But that would be quite inconvenient, especially in cases like batch jobs where the human user is no longer available to type her password. The Kerberos designers did not simply ignore the problem. Instead, each key has a version number. Network resources (including the KDC) should remember several versions of their own key. Since tickets expire in about 21 hours, there is no reason to remember a superseded key any longer than that. In tickets and other protocol messages, the key version number is sent, so that it can be known which key to use. Key version numbers are a little more problematic for humans. If Alice changes her password, the new value will be immediately reflected at the master replica of the KDC. If she subsequently attempts to log in before the change has been propagated to the slaves and her workstation contacts one of those slaves, the slave will return credentials encrypted with her old password. Her workstation will then fail to decrypt the credentials and so refuse to log her in. If Alice were as robust as the other processors supporting Kerberos, there would be no problem. The message would contain the key version number, and Alice would then know which password to supply. The Kerberos designers decided wisely not to introduce this piece of complexity into the user interface, but it does have the downside that for some period after a password change, the new password may fail. If the user tries the old password in such a situation, it may work, and the user may be a little confused. 11.9 ENCRYPTION FOR PRIVACY AND INTEGRITY Many of the data structures that Kerberos encrypts need to be protected from both disclosure and modification. In a ticket, for example, the KEY field needs to be protected from both disclosure and modification, and the NAME and EXPIRATION fields need to be protected from modification. Unfortunately, there is no standard mechanism for protecting both the confidentiality and integrity of a message with a single cryptographic pass. Generally the most useful way to protect the confidentiality of a message is to encrypt it in CBC mode (see §4.2.2 Cipher Block Chaining (CBC)). The most standard way to protect its integrity is with a CBC residue. But securely applying both of these techniques would require double the encryption effort and two separate keys (see §4.3.1 Ensuring Privacy and Integrity Together). Intuitively, it seems like it should be possible to make do with a single cryptographic pass by including some redundant information in the message to be encrypted and then to have the integrity of the message protected by having the recipient check that the redundancy is correctly done. A number of schemes have been proposed for doing this. Many have been found to have cryptographic weaknesses. None has gained general acceptance. The designers of Kerberos came up with their own variation on this theme, and subsequently cryptographic flaws were found with it. The 11.9 ENCRYPTION FOR PRIVACY AND INTEGRITY 281 flaws were not so egregious as to justify recalling Kerberos V4; however, a different approach is followed in Kerberos V5. One method for using DES on a long message is CBC. CBC does a good job on privacy. An intruder will not gain any information from analysis of the encrypted blocks. However, there is no integrity check that Kerberos can do to assure an application that the data was not tampered with. If an intruder were to modify block cn, then mn would be garbage, as would mn+1, but starting from mn+2 everything would decrypt properly. Kerberos would not be able to detect that mn and mn+1 had been garbled. Some applications might be able to tell, but Kerberos would like to provide the integrity assurance without depending on the application. Therefore, Kerberos did a modified version of CBC which they called Plaintext Cipher Block Chaining (PCBC). It is similar to CBC except that in addition to ⊕ing cn with mn+1, mn is also ⊕'d. In other words, in CBC, mn+1 ⊕ cn is encrypted to yield cn+1. In PCBC, mn+1 ⊕ cn ⊕ mn is encrypted to yield cn+1. PCBC has the property that modifying any ci will result in garbling of all decrypted plaintext blocks starting with mi all the way to the end. Kerberos puts some recognizable data at the end of a message that it will encrypt so that it can recognize whether the final block decrypts properly. It makes the assumption that if the final block decrypts properly, then the data has not been tampered with between the time it was transmitted by the source and received by the destination. However, using PCBC and checking the contents of the final decrypted block does not guarantee that Kerberos will be able to detect message corruption. For instance, if an intruder were to swap two adjacent blocks, PCBC will "get back in sync" after those two blocks, and the final block will decrypt properly (see Homework Problem 5). m1 m2 m3 m4 m5 m6 IV ⊕⊕⊕⊕⊕⊕ encrypt with secret key EEEEEE c1 c2 c3 c4 c5 c6 Figure 11-5. Plaintext Cipher Block Chaining (PCBC) 282 KERBEROS V4 11.10 11.10 ENCRYPTION FOR INTEGRITY ONLY Kerberos is often used by applications just for authentication. Kerberos also provides integrity protection and combined privacy and integrity protection for those applications that desire this facility. Combined privacy and integrity protection is provided by PCBC as described above. Integrity protection without privacy is described in this section. Kerberos V4 was designed with the expectation that DES encryption would be done in software, and that the processing cost of doing so would be a limiting factor in its acceptance. The designers were willing to use new or even cryptographically weak techniques in order to improve the performance. When integrity-protecting messages, the most straightforward and standard technique would have been to compute a DES CBC residue. Instead, they invented their own. It was based on a checksum algorithm designed by Jueneman [JUEN85]. Jueneman intended for his checksum to be used as a piece of redundant information inside an encrypted message, where it was less subject to cryptographic attack. Furthermore, the version implemented by Kerberos is substantially different (for instance, Jueneman did arithmetic based on mod 231−1, which Jueneman chose because it was prime, and Kerberos chose 263−1, which is not prime). The actual algorithm used by Kerberos V4 was never documented. In order to break it, or implement a compatible implementation, one would have to read the publicly available source code. No one has demonstrated the ability to break the Kerberos integrity checksum, perhaps because it has not been widely deployed for applications in which the payoff for breaking the scheme was worth the effort. So the fact that it never was broken is not strong evidence of its security. As with PCBC, it's a design upon which cryptographers would frown, but in practice it appears to have been good enough. The V4 checksum is computed on the session key concatenated with the message. Only the message and the checksum are transmitted—the session key is not sent. An intruder would not be able to compute a valid checksum on a modified message, since that would require knowledge of the session key. There are two possible problems with the use of the V4 checksum as a message digest function. One is that it might not be as cryptographically strong as a real message digest, and therefore it might be relatively easy to find a different message that yielded the same checksum, even without knowing the session key, since it might be possible to know how to modify a message in a way that would not change the value of the checksum. The second, and potentially more serious, problem is that it might be reversible. An intruder can see the entire plaintext message and the checksum. If it's possible to calculate the previous block, then perhaps it's possible to work backwards from the end and calculate the session key. Because of these possible weaknesses, this approach was abandoned in Kerberos V5. Instead, a choice of checksum algorithms is given, and in cases such as integrity-only, one of the standard message digest functions is used (see §12.8.1 Integrity-Only Algorithms). 11.11 NETWORK LAYER ADDRESSES IN TICKETS 283 11.11 NETWORK LAYER ADDRESSES IN TICKETS When Alice requests a TGT or a ticket to Bob, the KDC puts her 4-octet IPv4 address inside the ticket. Bob, when presented with a ticket, will check to make sure that the network layer source address on the connect request is the same as the one specified inside the ticket. Likewise, when the KDC receives a ticket request, it checks that the request is coming from the network layer address specified in the TGT. There are two reasons for putting Alice's network layer address in the ticket. The first reason is to prevent Alice from giving the ticket and session key to some third party, Ted, so that he can impersonate Alice. This restriction makes it more difficult for an intruder to walk up to the workstation Alice is using (when Alice steps out of her office), steal her session key and TGT, and use them from a safe location. But the restriction also prevents delegation, where Alice legitimately wants to allow something to act on her behalf. Indeed in Kerberos V5 delegation is specifically allowed (see §12.3 Delegation of Rights). The second reason is to prevent some third party Trudy from intercepting the ticket and authenticator on the wire and using it from Trudy's network layer address. Without Alice's network layer in the ticket, Trudy merely has to eavesdrop to obtain the ticket and an authenticator, use them within five minutes (or whatever the allowable clock skew is), and arrange for the authenticator as transmitted by Alice not to arrive. With Alice's network layer address in the ticket, Trudy has to additionally impersonate Alice's network layer address. This isn't necessarily difficult, but it adds one more hurdle for the attacker. It might be better to have Alice's network layer address be in the authenticator rather than the ticket. This allows the person using the ticket to pick the network layer address, which makes it easier to support a principal with multiple network layer addresses (Alice can choose which one to put into the authenticator), and it allows delegation. If Alice is willing for Ted to act on her behalf, and has therefore given him the session key and ticket, he can put his own network layer address into the authenticator. But Fred, who has not been given permission by Alice to act on her behalf, cannot generate an authenticator with his network layer address, so any security against eavesdroppers gained by having the network address in the ticket is equally applicable to having the network layer address in the authenticator. Putting the network layer address in the ticket instead of the authenticator is done to specifically disallow delegation. Kerberos V5 does allow delegation, but only with the mediation of the KDC and only if the originally acquired ticket explicitly permits it. We feel that since it is usually easy to forge a network layer source address, all the effort Kerberos has put into checking network layer source addresses is more trouble than it's worth. As with any higher-layer protocol that embeds layer 3 addresses as data, there are potential problems with NATs (see§15.2.1 NAT (Network Address Translation)) and migration to different address formats (e.g., IPv6). 284 KERBEROS V4 11.12 11.12 MESSAGE FORMATS In this section we'll go through the Kerberos V4 message formats. In all of the messages, we assume that Alice wants to talk to Bob. In Kerberos documentation Alice is sometimes referred to as the client and Bob as the server. We find that terminology confusing since both Alice and Bob are clients of the Kerberos system, and principals like file servers can indeed act in the role of client in terms of a Kerberos authentication. Therefore we will not use that terminology and will instead use Alice for the initiator of an authentication and Bob for the entity with which Alice would like to communicate, and the pronouns she for Alice and he for Bob. Fields that appear in packets: • NAME/INSTANCE/REALM of Alice or Bob—variable-length null-terminated character strings. • TIMESTAMP—a four-octet value representing the number of seconds since 00:00:00 GMT, 1 January, 1970. This is a common representation of time in UNIX-based systems. If treated as a signed integer, it can represent times between 1902 and 2038. If treated as an unsigned integer, it can represent times between 1970 and 2106. In Kerberos, some of the messages in which timestamps appear use the most significant bit as the D BIT (see below); in other messages, the most significant bit is set to zero. Therefore, Kerberos V4 effectively expires in 2038. (It's hoped that most people will have converted to V5 by then—or is that V19?). Kerberos V5 will not expire until December 31, 9999 (at least the timestamps will last till then). • D BIT, DIRECTION BIT—The purpose of this flag is to prevent an intruder from mirroring a message back to the sender (called a reflection attack) and having the sender accept it as a response. An asymmetry is created between the two ends of a conversation based on IP address. When the packet is traveling from the higher IP address to the lower IP address, the D BIT is set to 1. When the packet is traveling between two processes in a single node (so that both ends of the conversation have the same IP address), the TCP port number is used as a tie-breaker. By checking the D BIT, the receiver of a message can be assured that the message was generated by the other end of the connection. In the actual layout of the data, the D BIT is stolen from the high-order bit of the TIMESTAMP. Note that this is a dependency on IP, though one that could be adapted to most communication protocols. • LIFETIME—a one octet field, specified in units of 5 minutes. Therefore the maximum lifetime that can be expressed is a little over 21 hours. This limit has been seen as a major shortcoming of Kerberos V4, since it prevents giving tickets to long-running batch jobs. • 5-MILLISECOND TIMESTAMP—This field extends the TIMESTAMP field to give time granularity down to 5 milliseconds instead of 1 second. Its purpose is to allow more than one authenticator to the same service to be generated in a single second without having it rejected as a dupli- 11.12 MESSAGE FORMATS 285 cate. Actual time accuracy at that granularity is never needed. An acceptable implementation of this field would be a sequence number within the second. • PADDING—a field containing between zero and seven octets added to the end of any message that will be encrypted to ensure that the value to be encrypted is a multiple of eight octets. • NETWORK LAYER ADDRESS—a four-octet quantity which is an IPv4 address. • SESSION KEY—an eight-octet quantity which is used as a DES key. • KEY VERSION NUMBER—a one-octet quantity (see §11.8 Key Version Numbers). • KERBEROS VERSION NUMBER—a one-octet quantity equal to the constant 4. • MESSAGE TYPE—a one-octet quantity, with the low-order bit stolen for the B BIT (see next item). The high-order seven bits indicate one of the following message types: ♦ AS_REQ—Used when asking for the initial TGT. ♦ AS_REPLY (also TGS_REP)—Used to return a ticket, either a TGT or a ticket to some other principal. ♦ AP_REQ (also TGS_REQ)—Used to talk to another principal (or the TGS) using a ticket (or a TGT). ♦ AP_REQ_MUTUAL—This was intended to be used to talk to another principal and request mutual authentication. In fact, it is never used; instead, applications know whether mutual authentication is expected. ♦ AS_ERR—Used for the KDC to report why it can't return a ticket or TGT in response to AS_REQ or TGS_REQ. ♦ PRIV—This is a message that carries encrypted integrity-protected application data. ♦ SAFE—This is a message that carries integrity-protected application data. ♦ AP_ERR—Used by an application to report why authentication failed. Note: the names of the message types above do not match the names used in either the Kerberos V4 documentation or code. Because there was no consistent pattern, we used the names for the equivalent functions from Kerberos V5. • B BIT, BYTE-ORDER FLAG—This bit indicates whether fields holding four-octet integers have them in big-endian (most significant octet first) or little-endian (least significant octet first) order. Different machines have different native internal formats. Most protocols define a network byte order and force machines with a different native format to byte-swap whenever they send or receive integers. By use of this flag, Kerberos allows either order in transmitted packets, so in the common case where the native formats on the communicating machines are 286 KERBEROS V4 11.12.1 the same, no byte-swapping is necessary. This approach trades complexity for performance. If B BIT = 1, it means the least significant octet is in the lowest address in a multi-octet field. This is convenient for little-endian machines; big-endian machines like to put the most significant octet in the lowest address and are most happy with B BIT = 0. The following three data structures are never actually free-standing messages, but are instead pieces of information included in other messages. They are tickets, authenticators, and credentials. 11.12.1 Tickets A ticket for Bob is an encrypted piece of information that is given to principal Alice by the KDC and stored by Alice. It is encrypted by the KDC with Bob's master key. Alice cannot read what is inside, but she sends it to Bob who can decrypt it and thereby authenticate her. Now some comments about the fields which are not self-explanatory. • B BIT, BYTE-ORDER FLAG—LSB in a one-octet field, with the other bits unused. • ALICE'S NETWORK LAYER ADDRESS—four octets. Note that the field is fixed-length and four octets. Kerberos was designed to run in the network using the TCP/IP protocol suite. This fixed-length field means that this version of Kerberos cannot be used in a network with longer addresses. • TICKET LIFETIME. The units are 5 minutes, so the maximum ticket lifetime is a little over 21 hours. • TIMESTAMP—four octets. Time when ticket was created (the number of seconds since 00:00:00 GMT, 1 January, 1970). # octets 1 B ≤40 Alice's name null-terminated ≤40 Alice's instance null-terminated ≤40 Alice's realm null-terminated 4 Alice's Network Layer address 8 session key for Alice↔Bob 1 ticket lifetime, units of 5 minutes 4 KDC's timestamp when ticket made ≤40 Bob's name null-terminated ≤40 Bob's instance null-terminated ≤ 7 pad of 0s to make ticket length multiple of eight octets 11.12.2 MESSAGE FORMATS 287 • BOB'S NAME and INSTANCE. Including Bob's name and instance is of dubious value. There are two possible reasons for inclusion of this field in the ticket. One reason is that the field supports a node where several services share a key. If someone were to get a ticket to the gameplaying service on node DoD, and that service used the same key as the nuclear-missilelaunching service on the same node (it happened in War Games, right?), then having the name of the service in the ticket makes it impossible to use the ticket for the wrong service. For instance, suppose only once in a while (hopefully) does anyone ever use the nuclear-missile-launching service, but people often use the gameplaying service. Good guy A has rights to both services, and often uses the gameplaying service. One day bad guy C intercepts the ticket and authenticator that A was attempting to use to get to the gameplaying service, and then C quickly (within the clock skew) impersonates A's network layer address, substitutes as the destination socket number the nuclear-missile-launching service, and starts World War III. You wouldn't want that to happen now, would you? The other dubious purpose for this field is so that Kerberos can tell that the ticket decrypts properly. Kerberos V4 uses PCBC (see §11.9 Encryption for Privacy and Integrity), and therefore assumes that any tampering with the data would result in the final block not decrypting properly. Since the final blocks include Bob's name and instance, Bob will be able to recognize if they are correct. (This is not a strong reason for the field because, even if PCBC worked as hoped, there is no value to be gained from tampering with a ticket and turning the information inside to garbage. User data is a different